UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: ALGOLIPSE: un visualizador de algoritmos y estructuras de datos para Eclipse
Autores: Falcone Facundo, Ronconi Lisandro
Director: Fava Laura
Codirector: Rosso Jorge
Asesor profesional:
Carrera: Licenciatura en Sistema y Licenciatura en Informática

Resumen

En este trabajo, se presenta una extensión para Eclipse que genera automáticamente visualizaciones de estructuras de datos y algoritmos a partir de código JAVA implementado por los estudiantes y no intervenido. Esta propuesta ayuda a reducir la abstracción que conlleva el estudio de estos temas sin el esfuerzo extra que implica aprender una nueva herramienta, lenguaje o librerías especiales para la visualización de algoritmos.

Palabras Claves

Plugin para Eclipse, Standard Widget Toolkit (SWT), Algoritmos y Estructura de Datos, Programación Orientada a Aspectos (POA), AspectJ.

Trabajos Realizados

Investigación sobre nuevas herramientas de visualización de estructuras de datos y algoritmos asociados, su utilización y comportamiento.
Desarrollo de un plugin para la visualización de la estructura de datos usando Programación Orientada a Aspectos.

Conclusiones

Después de haber investigado sobre la problemática con la que se encontraban los estudiantes a la hora de entender correctamente los algoritmos recursivos, se ha propuesto y desarrollado un plugin para Eclipse que permita ver, seguir y entender dichos algoritmos. Se ha investigado sobre diferentes herramientas tales como Visualgo, JGraph, etc. que sirvieron de guía para la realización del proyecto.

Trabajos Futuros

Incorporación de nuevas estructuras de datos al Plug-in para Eclipse.
Desarrollo de una interfaz de usuario gráfica más sofisticada para la visualización de las estructuras y animaciones.

Agradecimientos

Facundo

A mi madre

A mi mama por haberme apoyado en todo momento, por sus consejos, sus valores, por la motivación constante que me ha permitido ser una persona de bien, pero más que nada, por su cariño.

A mi padre

A mi papa por los ejemplos de perseverancia y constancia que lo caracterizan y que me ha infundado siempre, por el valor mostrado para salir adelante y por su cariño.

A mis abuelos

Mis abuelos (Q.E.P.D) por quererme y apoyarme siempre, esto también se lo debo a ustedes.

A mis hermanos

Fernando y Agustina, por estar conmigo y aconsejarme en todo momento.

A mi novia

A Amalia por su apoyo y ánimo que me brinda día a día para alcanzar nuevas metas, tanto profesionales como personales.

A mis familiares

A mis tíos Sandra y Ricardo, a mis primos Sebastián, Juan Martin y Manuel, y a todos aquellos que participaron directa o indirectamente en la elaboración de esta tesis.

A mis profesores

A mi tutora, Laura Fava y codirector, Jorge Rosso por guiarnos en la tesina, brindándonos sus conocimientos. Por apoyarnos y motivarnos para la culminación de nuestros estudios profesionales y por la elaboración de esta tesina.

A la Familia Tamagno

Por dejarme ser parte de su familia. En especial a Walter por enseñarme a no bajar los brazos.

A mis amigos

Por confiar y creer en mí y haber hecho de mi etapa universitaria un trayecto de vivencias que nunca olvidaré.

Lisandro

Primero y más importante quiero agradecer a mis padres que me ayudaron e impulsaron durante todos los años de estudio, brindándome conocimientos y buenos concejos.

Por segundo lugar a mi hermano que me ayudo en los estudios y en buenas practicas.

Quiero agradecer también a mis amigos, que me brindaron su apoyo e impulsaron a seguir adelante.

Voy a dar un especial agradecimiento a mi primo Francisco que me aconsejo y ayudo en el mundo universitario, también por haberme abierto las puertas al mundo laboral

A mi tutora, Laura Fava y codirector Jorge Rosso, por darme la oportunidad de aprender con ellos y de poder lleva a cabo la tesina; brindándome sus conocimientos, ideas y buenas practicas.

Índice

Capítulo 1 - Introducción y Motivación

1.1. Motivación

Las estructuras de datos, son un camino para almacenar y organizar los datos con un cierto orden y de esa manera facilitar el acceso y modificación de los mismos [1]. No todas las estructuras cubren el amplio espectro de problemas que pueden llegar a surgir en la actualidad; es por esto que a la hora de realizar un algoritmo se debe pensar en cuál es la mejor solución para tal caso. Tal vez este es el punto más importante. Representar la información es fundamental en ciencias de la computación y en informática. De modo práctico, los requisitos de almacenamiento y tiempo de ejecución exigen que tales programas deban organizar su información de un modo que soporte procesamiento eficiente. Por estas razones, el estudio de estructuras de datos y de los algoritmos que las manipulan constituye uno de los puntos más importantes en esta área. La algorítmica es un campo primordial en el procesamiento de datos. Sin embargo, ciertos conceptos asociados a estos temas no siempre resultan simples de comprender para los alumnos. Knuth recomienda que la mejor forma de aprender y entender un algoritmo es probarlo mediante un ejemplo [2].

Las experiencias propias como la de otros alumnos, nos llevan a interpretar que el desarrollo de un algoritmo eficiente en cuanto a la estructura elegida resulta complicado; más aún aquellas estructuras que por eficiencia necesitan ser recorridas mediante la recursividad. Un concepto importante es que en muchos problemas no basta con escribir programas que funcionen. Si el programa va a utilizar grandes cantidades de datos, entonces el tiempo de ejecución se vuelve un problema.

Esta tesina girará en torno a la creación de un plug-in para Eclipse denominado “Algolipse”. Tal herramienta centrará su atención en la visualización interactiva de los algoritmos, realizados por los alumnos de segundo año de la Facultad de Informática e Ingeniería en Computación. Se pensó en Eclipse, debido a que es la herramienta utilizada por los mismos. Si bien es posible realizar un seguimiento a través del “debugger”, no siempre resulta intuitivo para personas con poca experiencia en la herramienta; por lo que terminan ejecutando el código y realizando un procedimiento de “prueba y error”, que lleva a programas complicados, poco eficientes, difíciles de defender y entender. Tal vez este último punto es el que más nos inspiró para tal suceso.

La herramienta persigue además fomentar el trabajo continuado y personal de los alumnos tratando que la misma les sea de apoyo y motivación. De esta manera ayudarlos a afrontar la cursada lo mejor posible.

1.2. Objetivos y Metodologías a Emplear

El objetivo de esta tesina es diseñar e implementar una herramienta que permita seguir la ejecución de algoritmos visualizando sus estructuras de datos y sus algoritmos de acceso.

Esta herramienta está orientada a alumnos de la cátedra de Algoritmos y Estructuras de Datos y de Programación 3 de la Facultad de Informática y podría ser usada como un complemento novedoso de las clases teóricas y prácticas, en las que el alumno incorpora conceptos y los aplica en el desarrollo de sus propios algoritmos. El uso de una herramienta de soporte para la enseñanza/aprendizaje que visualice las diferentes estructuras de datos y el comportamiento de los algoritmos sobre ellas resulta de suma importancia, especialmente para los estudiantes de los primeros años de las carreras de informática, que no están totalmente familiarizados con estos temas. Se investigó sobre algunas herramientas tales como JGrasp, Visualgo, las cuales no cumplen con todos los objetivos planteados, pero nos han servido de base para comenzar el desarrollo de nuestra herramienta.

En esta tesina se analizarán los criterios que encaminaron al desarrollo de "*Algolipse",* las distintas herramientas que han facilitado su creación, los aspectos técnicos de implementación y otros puntos importantes que fueron útiles para desarrollar la herramienta.

Las distintas fases realizadas, fueron pensadas para que sea lo más escalable posible y de esta manera, las críticas constructivas que se reciban por parte de los alumnos como de los profesores, puedan ser plasmadas sin ningún inconveniente.

1.3. Estructura de la Tesina

Habiendo mencionado la idea general de esta tesina, continuaremos con el segundo capítulo explicando el Estado del Arte para poder realizar una comparativa entre lo que en la actualidad se encuentra funcionando y lo que nosotros queremos mejorar. Para tal suceso, se explicarán alguna de las herramientas existentes denotando sus ventajas y desventajas como también explicando el uso de las mismas.

Siguiendo con el capítulo tres, nos centraremos en manifestar la descripción del problema. Se realizó un análisis de los principales inconvenientes con los que se

encontraban los alumnos en la actualidad, como también tomando nuestras propias experiencias al respecto.

Una vez finalizado, abordaremos el capítulo cuatro donde se establecerá la Propuesta de Solución, explicando que es lo que se pretende realizar para mejorar el estado del arte.

Proseguiremos con el capítulo cinco detallando las diversas herramientas que fueron utilizadas como también el uso que se les está dando en nuestro desarrollo.

Se continuará con el capítulo seis viendo en detalle la creación de un proyecto plugin junto a los pasos que se deben seguir para configurarlo y las diferentes opciones que este nos ofrece para agregar nuevas características.

En el capítulo siete se definirán tanto los modelos UML como de secuencia que servirán para el desarrollo de la herramienta; explicando los componentes más importantes de los mismos. A su vez se explicará la arquitectura de la herramienta.

Debido a que gran parte del desarrollo utiliza interfaces gráficas, en el capítulo ocho se explicará la herramienta para tal suceso, dando ejemplos y mostrando su funcionamiento.

Terminando el episodio anterior, seguiremos con el capítulo nueve comentando la puesta en marcha del plugin por parte del alumno.

Al ser uno de los puntos más importantes del proyecto, se realizará en el capítulo diez un apartado sobre Programación Orientada a Aspectos, explicando sus características, las distintas formas de aplicarlo como también algunos ejemplos para que se pueda entender mejor.

Antes de finalizar, en el capítulo once, se mostrará el funcionamiento de la aplicación a través de ciertas capturas que fueron tomadas de pruebas realizadas.

Finalizamos los capítulos con el número doce, donde se presentará la conclusión de esta tesina como también los trabajos que quedan por realizar.

Capítulo 2 - Estado del Arte

En la actualidad existen algunas herramientas que permiten realizar simulaciones de estructuras de datos de una manera muy precisa pero que no son lo suficientemente completas, o no dan la posibilidad de ser integradas al ambiente de trabajo utilizado por los alumnos.

2.1. Visualgo

Es un sitio web desarrollado por un equipo de profesores y estudiantes de la Facultad de Informática de la Universidad Nacional de Singapur. Visualgo, visualiza una serie de estructuras de datos y algoritmos que permite entender el comportamiento de las estructuras cuando se le solicitan diferentes operaciones. Es un software interactivo y web para la visualización de algoritmos clásicos. Es interactivo en el sentido que los estudiantes pueden ingresar datos y ejecutar los algoritmos predefinidos. La pantalla principal muestra un conjunto de algoritmos que pueden ser ejecutados e interactuados. Durante la ejecución se despliega un panel que visualiza un script predefinido y fijo con la lógica del mismo. Visualgo permite visualizar el manejo de las estructuras básicas y algoritmos asociados a través de una interfaz de usuario gráfica, moderna y unificada, que incluye desde algoritmos básicos de manejo de listas hasta algoritmos complejos vinculados con grafos.

La página principal, cuenta con una serie de estructuras donde se puede realizar la elección de la misma.

Figura 1 Visualgo

Una vez realizada la selección, nos exhibe una nueva pantalla junto a un ejemplo de la estructura seleccionada. En la parte inferior izquierda permite seleccionar alguna de las operaciones a realizar.

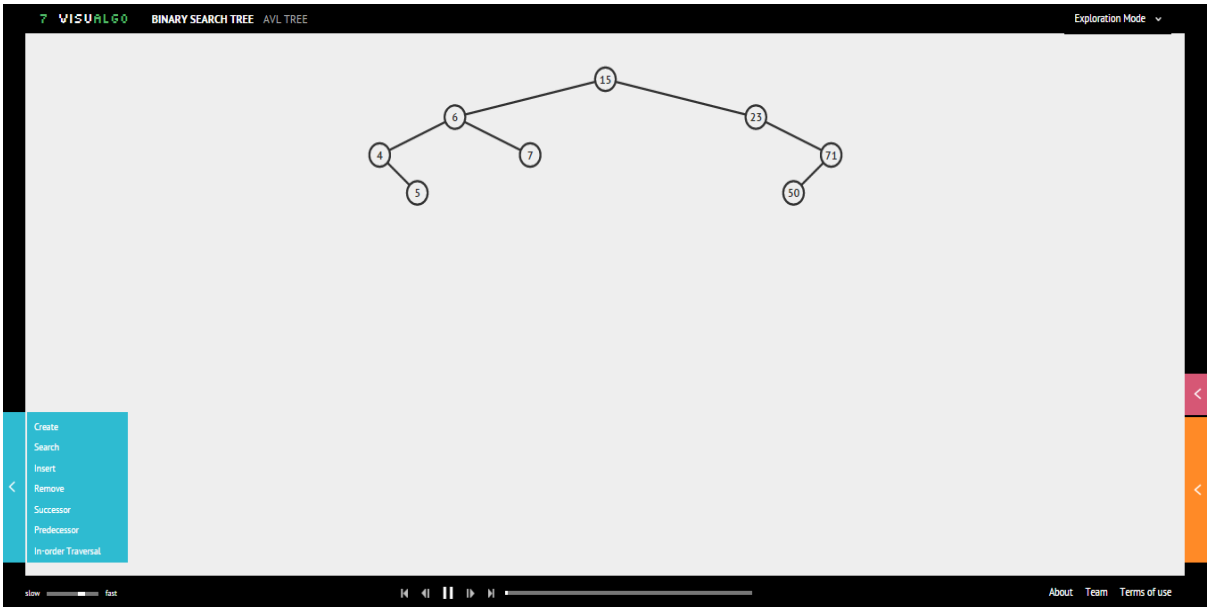

Figura 2 Arbol en Visualgo

Como ejemplo, en la siguiente imagen se muestra la inserción del nodo 9 en una estructura Árbol Binario de Búsqueda. La esquina inferior derecha se encarga de mostrar el algoritmo asociado a la operación seleccionada.

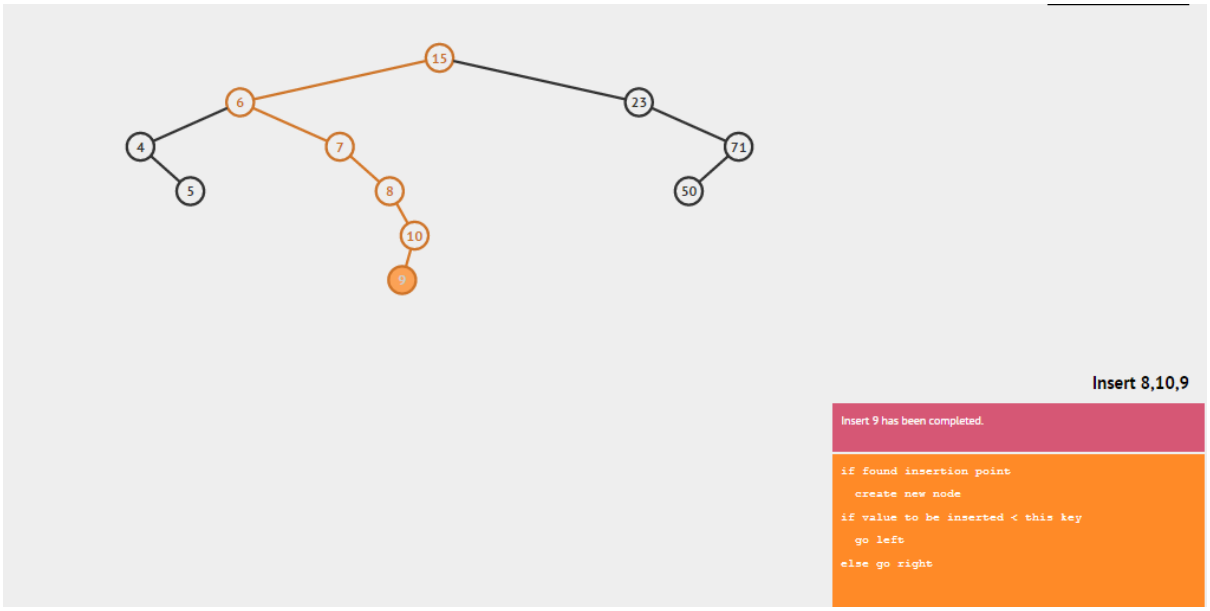

Figura 3 Recorrido en visualgo

2.2. JGrasp

jGRASP [3] es un entorno de desarrollo ligero, creado específicamente para proporcionar la generación automática de visualizaciones de software que permiten la comprensibilidad del mismo. Está realizado en el Lenguaje de Programación Java por Gaudenz Alder como un proyecto universitario en el año 2000.

La implementación de JGrasp se encuentra enteramente basada en las clases JTree. No es una extensión sino una modificación de su código fuente utilizando componentes Swing para realizar la visualización de los árboles. Los componentes soportan ventanas extendidas, opciones editables y su semántica está definida por la aplicación. Por lo tanto JGrasp es altamente adaptable.

JGrasp permite la transferencia de un modelo junto a una descripción de su estructura, gráfico, y de su patrón geométrico. Basados en los componentes de Swing, nos provee de mapas para describir y cambiar las celdas de un modelo.

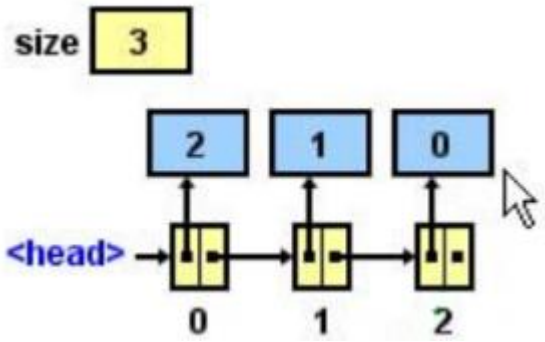

Figura 4 Ejemplo de una visualización en JGrasp

2.3. JavaMy

Esta herramienta provee una completa visualización de las estructuras de datos más usadas, como arreglos, pilas, colas, árboles y grafos y la animación de las operaciones más comunes sobre ellas [4]. A su vez provee una animación de algoritmos simples definidos por el usuario, a través del uso de un lenguaje propio llamado javaMy que es muy similar a Java.

Para la visualización de las estructuras, el usuario debe instanciar aquellas estructuras que quiera observar utilizando el tipo de dato Observable proporcionado por el software. A partir de este momento, un frame de animación es creado para que las estructuras que hayamos instanciado se agreguen. De esta manera, a medida que se vaya ejecutando el algoritmo, se van a ir viendo los cambios que va sufriendo la estructura.

Si bien nos da la posibilidad de interactuar con una gran variedad de estructuras de datos, tiene la desventaja que utiliza un lenguaje de programación distinto al utilizado por los alumnos.

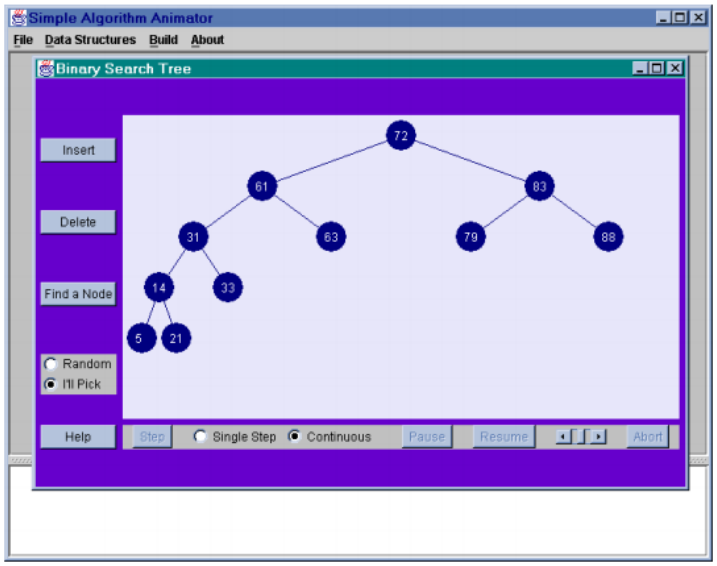

Figura 5 Ejemplo de visualización de un árbol en JavaMy

2.4. CADILAG

Está siendo desarrollada en la Universidad Estadual Paulista, en Presidente Prudente, Brasil. Fue desarrollada en JavaFX, permitiendo que sea utilizada como aplicación de escritorio o basada en web [5]. Ofrece un conjunto limitado de estructuras de datos con las cuales trabajar, y a partir de la selección de una de ellas, la herramienta permite visualizar el comportamiento de sus operaciones básicas. Contiene una ayuda general con explicaciones sobre las operaciones y una asistencia para entender la estructura estudiada al momento de visualizar la animación. Si bien, es posible interactuar de manera amigable con las estructuras de datos y comprender cómo funcionan las operaciones, no es posible que los estudiantes verifiquen el funcionamiento de su propio código

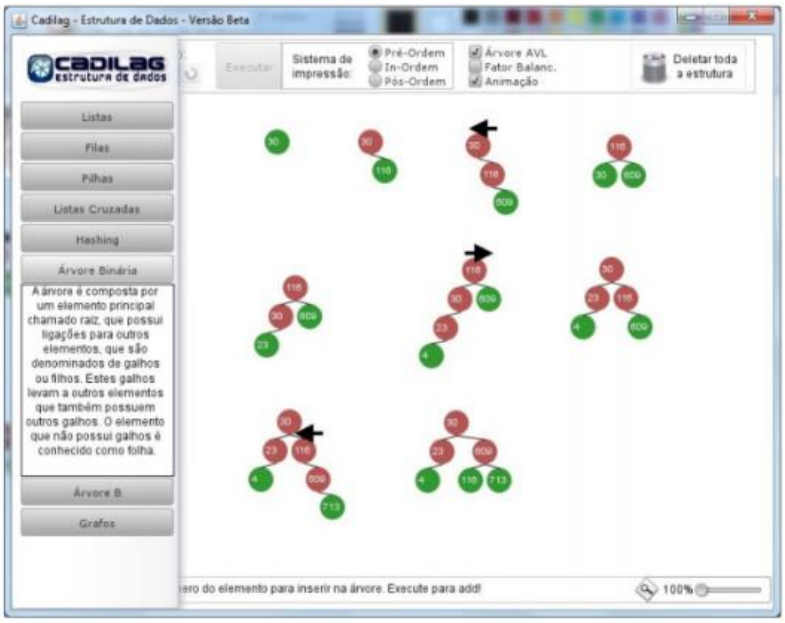

Figura 6 Ejemplo de una visualización en Cadillag

Capítulo 3 - Descripción del Problema

Los temas relacionados con la algorítmica y el uso de las estructuras de datos aparecen en asignaturas de los primeros años de las carreras de Ciencias de la Computación, lo que aumenta la complejidad de su enseñanza-aprendizaje. En particular, en la Facultad de Informática de la Universidad Nacional de La Plata, la materia que comprende estos contenidos es Algoritmos y Estructuras de Datos, asignatura obligatoria que corresponde a segundo año de las carreras de grado: Licenciatura en Informática, Licenciatura en Sistemas y Analista Programador Universitario. Además, en segundo año de la carrera de Ingeniería en Computación se dicta la materia Programación 3, también de carácter obligatorio, que incluye los temas mencionados. En estas asignaturas se abordan los temas relacionados con el uso y aplicación de estructuras de datos básicas y avanzadas, algoritmos que permiten la manipulación de dichas estructuras y también el análisis de la eficiencia de los mismos.

Las diversas opiniones, el día a día en la Facultad, como así también nuestras propias experiencias, nos llevan a comprender la problemática con la que se encuentran los alumnos a la hora de poder entender los algoritmos desarrollados por ellos mismos que por alguna razón son más eficientes que otros; teniendo mayor énfasis los recursivos. No siempre un algoritmo funcionando es lo suficientemente claro y simple; siendo esto una característica de vital importancia para el desarrollo que nos encontremos realizando, más aún en aquellos que requieren grandes cantidades de datos y las búsquedas de los mismos deben realizarse lo más simple posible. Por tal motivo, lo importante es que el alumno sea capaz de reconocer y elegir el algoritmo más adecuado. Un buen programador buscará el algoritmo más óptimo dentro del conjunto de aquellos que resuelven con exactitud un problema dado.

Ciertas operaciones realizadas como el “buscar, insertar, eliminar”, en estructuras de datos como Árboles Binarios, suelen ser más simples y claras si se realizan de forma recursiva. El proceso de entender dichos procedimientos, se torna engorroso sobre todo cuando se viene con una idea de programación donde la manera iterativa y secuencial son “habitúes”. Si bien el ambiente de desarrollo utilizado nos provee de un “debug”, en un principio no estamos lo suficientemente capacitados para poder utilizarlo de manera correcta por lo que se deja de usar y es en ese momento donde caemos en el inevitable “prueba y error”.

No obstante que la recursión es un problema primordial, es necesario entender cuándo usarla sin que termine perjudicando el algoritmo. Es decir, hay casos donde la

utilización de la forma iterativa es más adecuada. Por ejemplo, el recorrido de una lista o una cola. Hemos visto que hay casos donde el alumno se encuentra con la problemática de saber elegir el algoritmo correcto.

Capítulo 4 - Propuesta de Solución

Habiendo mencionado la problemática, nos centraremos en dar la solución de la misma.

Anteriormente, explicamos los inconvenientes con los que se encontraban los alumnos a la hora de utilizar el *debug* de Eclipse por primera vez y las consecuencias que traía. A su vez habiendo investigado en el estado del arte, no encontramos herramientas que posean la capacidad de realizar una especie de seguimiento de la recursión para ver donde el algoritmo se encuentra detenido en cierto momento y donde se encontraba en instancias anteriores. Es por esto que nos planteamos en realizar una herramienta, más específicamente un plugin [6], que permita realizar un seguimiento de los algoritmos que ellos mismos realicen de una forma interactiva, para que les sean más legibles, explicativos y principalmente, que pueda lograr el objetivo principal de comprender lo que ellos confeccionan y el funcionamiento de la recursión. Si bien en una primera versión, la misma contendrá a los Árboles Binarios, Árboles Binarios de Búsquedas, Árboles Generales y Listas, está pensada para que pueda crecer e incluir nuevas estructuras (Como Grafos).

Debido a que la idea era tratar de asemejarse a lo que hace el "*debug"*, una vez seleccionado y ejecutado el procedimiento a inspeccionar, la aplicación se detendrá al encontrar un llamado del mismo mostrando gráficamente el estado actual de la estructura junto a cierta información, como por ejemplo lo que retorna el algoritmo, y volverá a comenzar luego de que se presione sobre continuar; de esta manera se irá recorriendo la estructura a través de un "paso a paso". Si el algoritmo finaliza de forma correcta, se mostrará un mensaje de Éxito; pero si se produce algún error, el plug-in lo denotará con un mensaje de alerta donde se podrá ver la excepción. Conjuntamente, se dará la opción de acceder a una ventana de "logs", donde se irá guardando un historial de las llamadas recursivas a dicho procedimiento.

Para poder utilizar el plugin, los alumnos tendrán que importarlo en su Eclipse (*Ver Capítulo 9 Puesta en marcha*) y a través de una ventana configurarlo.

Capítulo 5 - Herramientas y Metodologías Utilizadas

5.1. Herramientas de ayuda

Trello

Para llevar a cabo el desarrollo, se utilizó una herramienta simuladora de "*Scrum*" conocida como Trello. Como primera medida, se realizó una clasificación entre cinco componentes los cuales son:

1. Backlog. Tareas que fueron mencionadas para, en un futuro, validar su posible implementación.
2. toDo. Tareas por completar.
3. inProgress. Tareas que comenzaron a realizarse pero no fueron finalizadas.
4. done. Tareas que fueron finalizadas.
5. problems. Problemas que fueron encontrados y hasta el momento no hay solución.

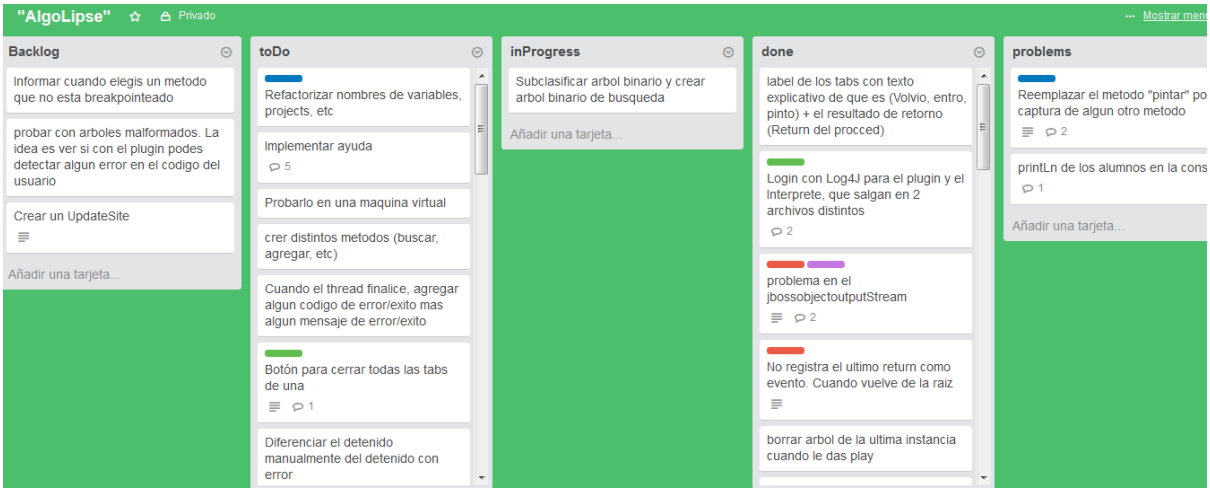

Figura 7 Trello

La gran ventaja que posee "*Trello*" es la facilidad de comunicación que hay entre los desarrolladores ya que no es necesario enviar un correo para comunicar lo que se está realizando o ver qué tareas tiene asignado cada uno. Además, permite llevar un control sobre las tareas que quedan pendientes y las tareas que van siendo realizadas y el tiempo que estas costaron.

Git

Para poder integrar de manera correcta el código realizado, sin que haya problemas de versionado, se pensó en un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Un sistema de control de versiones permite la creación de una historia para una colección de archivos e incluye la funcionalidad para revertir la colección de archivos a otro estado. Esta colección de archivos generalmente es llamada "código fuente". En un sistema de control de versiones distribuido todos tienen una copia completa del código fuente (incluyendo la historia completa del mismo) y puede realizar operaciones referidas al control de versiones mediante esa copia local.

Git mantiene todas las versiones. Por lo tanto, se puede revertir a cualquier punto en la historia de tu código fuente. A su vez, posee ciertas características como clonar un repositorio existente e ir sincronizando los cambios (sentencia push), como también obtener los cambios desde un repositorio remoto (sentencia pull).

5.2. Herramientas para la creación

Sockets

Un socket es un enlace de comunicación bidireccional entre programas ejecutando sobre la red [7]. Hay varios tipos de socket, siendo los más usados: Stream Sockets (Socket orientados a conexión, usando TCP) y Datagram Sockets (Sockets sin conexión, usando UDP).

La principal diferencia entre ambos se debe a que uno es orientado a la conexión y otro no. Los Stream Sockets, al utilizar TCP como protocolo de comunicación, en primer lugar, hay que establecer una conexión entre los sockets. Mientras uno atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

Por otro lado, los Datagram Sockets al utilizar UDP como protocolo de transporte, se los denomina sin conexión. Se transmiten paquetes individuales de información y no garantiza que los paquetes lleguen en alguna forma en particular. De hecho, los paquetes pueden perderse, duplicarse e incluso llegar en desorden.

Java provee clases bajo el paquete “java.net” para facilitar y abstraer la conexión. Estas clases son Socket (Que representa un socket) y ServerSocket (Que le provee un

mecanismo al servidor para escuchar conexiones nuevas), las cuales usan TCP (Stream Socket) para comunicarse.

Los siguientes pasos ocurren cuando se establece la comunicación:

- El servidor instancia un objeto de clase ServerSocket, indicando el puerto en el cual escuchará.
- El servidor envía el mensaje *accept()* al objeto ServerSocket. Este mensaje bloquea al proceso hasta que un cliente se conecte.
- El cliente inicializa un objeto de clase Socket, especificando la dirección del servidor y el puerto.
- El constructor del Socket intenta conectar el cliente al servidor en la dirección y puerto especificados. Si la conexión es exitosa, el cliente tiene un objeto Socket capaz de comunicarse con el servidor
- En el lado del servidor, el mensaje *accept()* devuelve una referencia a un nuevo Socket que está conectado al socket del cliente

Luego de establecer la conexión, se pueden realizar comunicaciones bidireccionales usando Stream de entrada y salida. Los sockets poseen un InputStream y OutputStream (El InputStream del cliente se conecta al OutputStream del servidor). En el desarrollo se recurrió a una librería externa llamada JBossSerialization, para serializar objetos en la comunicación entre ambos sockets y evitar que el código de los alumnos sea manipulado por ellos mismo agregando la implementación Serializable. Dichas librerías aportan clases que facilitan la serialización de objetos Java.

Clientes y Servidores

Desde el punto de vista funcional, se puede definir la computación Cliente/Servidor como una arquitectura distribuida que permite a los usuarios finales obtener acceso a la información en forma transparente aún en entornos multiplataforma. En el modelo cliente servidor, el cliente envía un mensaje solicitando un determinado servicio a un servidor (hace una petición), y este envía uno o varios mensajes con la respuesta (provee el servicio). En un sistema distribuido cada máquina puede cumplir el rol de servidor para algunas tareas y el rol de cliente para otras.

En nuestro desarrollo, el Cliente está representado por un thread (ListenerThread), que se levanta por el plugin cuando se presiona sobre el botón “play” (La idea del thread,

está centrada en controlar la comunicación con el servidor de forma asincrónica al plugin). Lo primero que realiza es crear un proceso donde corre el servidor (línea 54), y generar un socket por el cual está esperando la conexión en un puerto específico (línea 81). Una vez aceptada la conexión, se envían al Servidor los parámetros necesarios para el correcto funcionamiento de la herramienta (línea 83 - 86).

ListenerThread (Cliente)

```
44      public void run(){
45          try {
46              log.info(this.getName()+ " Arrancado");
47              setThreadState( ThreadState.RUNNING );
48              getExecuteManager().getModel().getEventHistory().setLastEvent("**********Comenzand
49              setPriority(MIN_PRIORITY);//para que al obtener el acceso exclusivo a aspectMonito
50
51              //Crea el proceso de interprete
52              String pathInterprete = System.getProperty("user.home")+"/interprete.jar";
53              log.info("Creando proceso interprete: Path del interprete: "+pathInterprete );
54              ProcessBuilder processBuilder = new ProcessBuilder(
55                      "java",
56                      "-jar",
57                      pathInterprete);
58
59
60              process = processBuilder.start();
```

Figura 8 ListenerThread - Inicia el proceso interprete

```
81              communicator.init(5000);
82
83              communicator.storeObject(PathClass.PATH_USER_PROYECT);
84              communicator.storeObject( getMethodsBreakpointNames() ); //Se envian solo los nombre de los metodos
85              communicator.storeObject( getExecuteManager().getExecuteConfiguration().getMainClass() );
86              communicator.storeObject( getExecuteManager().getExecuteConfiguration().getVariableArbol() );
```

Figura 9 Inicio de la comunicacion con el Interprete

En el momento que el proceso que representa al servidor comienza, abre el socket y queda a la espera de la conexión con el Cliente. Ya establecido el enlace, el Servidor recibe los parámetros enviados por el mismo (línea 28 - 35).

Intérprete (Servidor como proceso del Sistema Operativo)

```
15 public class Interprete {
16
17     private static Logger log = Logger.getLogger(Interprete.class);
18     protected static Communicator communicator = ServerCommunicator.getCommunicator();
19
20     public static void main(String[] args){
21         try{
22             log.debug("Interprete inicia");
23             communicator.init( 5000 );
24
25             log.debug("Conexion establecida");
26             //recibir por el puerto el path del proyecto, Lista de methodos a breakpointear, Main class
27
28             String pathProject =(String) communicator.readObject();
29             log.debug("Path:"+pathProject);
30             List<String> methodsName = (List<String>) communicator.readObject( );
31             log.debug("methodName: "+methodsName);
32             String mainClassName = (String)communicator.readObject( );//"nuevo.Main"
33             log.debug("mainClassName: "+mainClassName);
34             String variableArbol = (String)communicator.readObject( );
35             log.debug("Variable arbol: "+variableArbol);
```

Figura 10 Intérprete - Inicio de la comunicación con el cliente

Este intérprete se ejecuta como un proceso (Del S.O) aparte, ¿Por qué se pensó así? Pensemos en que los alumnos realizan sus propios programas y los prueban con la herramienta (La herramienta posee su propio “play” y “stop”). ¿Qué sucedería si por alguna razón sus implementaciones caen en un loop infinito? ¿Habría forma de detectar dicho suceso y frenarlo sin necesidad de reiniciar el Eclipse? Justamente es aquí donde pensamos en este modelo. Al establecer al Servidor como un proceso, podremos terminarlo en cualquier momento enviando la señal “kill”; de esta manera con solo implementar un botón “stop” que efectúe dicha señal podremos detenerlo sin necesidad de reiniciar el Eclipse.

Programación Orientada a Aspectos (POA)

Los continuos avances en la ingeniería del software han ido incrementando la capacidad de los desarrolladores para descomponer un sistema en módulos independientes cada uno con una función bien definida, esto es, facilitar la separación de intereses (separation of concerns, SOC). Gracias a la POA se pueden encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos [8] [9] [10]. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables.

La separación de intereses (separation of concerns) consiste en descomponer un sistema en módulos independientes de forma que cada uno de ellos realice una función específica.

En el capítulo 10 se abordará con más detalle este tema, como también se explicará el uso en nuestro desarrollo.

Reflection

Es la habilidad que poseen los programas para que se examinen a sí mismo como también manipular sus propiedades internas. Ciertos lenguajes como Pascal, C o C++ no tienen la posibilidad de obtener información acerca de las funciones definidas dentro de sus programas. Un uso tangible de esta tecnología se puede ver en los "JavaBeans[1]" donde los componentes de software pueden ser manipulados visualmente a través de una herramienta de construcción. Dicha herramienta utiliza Reflection para obtener las propiedades de las clases a medida que son cargadas dinámicamente.

Configuración para utilizar Reflection

A modo de ejemplo, se mostrarán los pasos necesarios para obtener un objeto de la clase java.lang, para pedirle sus métodos e imprimirlos [11].

1. El primer paso consiste en obtener un objeto *java.lang.Class,* de la clase que queremos manipular.

2. El segundo paso es llamar a un método que nos devuelva la lista de todos los métodos declarados por la clase (getDeclaredMethods).

3. Una vez obtenida esta información, el tercer paso es utilizar la API Reflection para manipularla.

```
Class c= Class.forName("java.lang.String");
Method m[] = c.getDeclaredMethods();
System.out.println(m[0].toString());
```

En nuestro desarrollo, el uso de Reflection está dado debido a que el plugin necesita recorrer clases, variables y métodos que están definidos por el alumno (Así como también ejecutar dicho código) y no son conocidos en tiempo de compilación.

La tarea del plugin es ejecutar código que pertenece al alumno. Esta ejecución debe ser previamente configurada (Main class, clase Árbol, métodos de la clase Árbol, etc). Reflection se utiliza para obtener del proyecto, las clases que definió, a partir del cual podrá especificar cuál es la clase Main y cuál es la clase que representa la estructura a recorrer. A

[1] Modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones en Java. Se usan para encapsular varios objetos en un único objeto, para hacer uso de un solo objeto en lugar de varios más simples

su vez, a partir de dicha clase, se debe especificar los métodos que se van a utilizar para recorrer la estructura, para lo cual se necesita obtener todos los métodos públicos de la clase en cuestión. El alumno debe a su vez, especificar cuál es el objeto a inspeccionar, por lo que se debe obtener las variables de la clase Main.

Para la visualización de la estructura del alumno, la cual se recibe como un objeto del que desconocemos su tipo y no puede ser casteada a ninguna clase conocida, se pensó en desarrollar una estructura similar, de manera que se vaya construyendo a partir de los parámetros capturados. Para tal suceso, también se utilizó Reflection.

```
44          /**
45           * Obtenemos el metodo getValue concreto de esa clase
46           */
47          Method method = arbol.getClass().getMethod(
48                  executeConfiguration.getMethodValue().getName(), null);
49          Object value = method.invoke(arbol, new Object[0]);
```

Figura 11 obtener el getValue()

Como podemos observar, en la línea 47, obtenemos el objeto Method que representa el getValue() del nodo. Finalmente, en la línea 49, este método se invoca sobre el árbol para obtener el valor del nodo.

```
77              /**
78               * Buscamos el hijo derecho del objeto arbol
79               */
80              method = arbol.getClass()
81                      .getMethod(
82                              executeConfiguration.getMethodHijoDerecho()
83                                      .getName(), null);
84              Object hijoDerecho = method.invoke(arbol, null);
85              if (hijoDerecho != null) {
86                  ArbolBinario<Object> subArbolDer = transformerNode(hijoDerecho,
87                          actualData, invokedMethod, executeConfiguration);
88                  subArbol.agregarHijoDerecho(subArbolDer);
89              }
```

Figura 12 Buscar hijo derecho

En este fragmento de código, podemos ver un comportamiento similar al anterior, pero esta vez para obtener el subárbol derecho del árbol. En la línea 80, se obtiene el objeto Method, que representa el *getHijoDerecho()* del mismo, el cual se invoca en la 84 (En la sentencia “if” de la línea 85, se valida que el subárbol no sea nulo, ya que se realiza de forma recursiva). El método para obtener el subárbol izquierdo, se realiza de la misma manera.

Otro uso importante de la herramienta, se tiene a la hora de ejecutar el plugin. La ejecución del programa del alumno se lleva a cabo cargando la clase Main y ejecutando el

método *public void main(String[] args)* (Método invoke del objeto Class), todo esto utilizando Reflection.

```
42              Class[] argTypes = new Class[] { String[].class };
43              Method mainMetodo = rClassLoader.getMainClass().getDeclaredMethod("main", argTypes);
44              String[] mainArgs = {};
45
46              try{
47                  log.debug("Arranca mainMethod .... Invoke");
48                  mainMetodo.invoke(null,(Object) mainArgs);
49                  log.debug("Invoke terminado");
50                  CommunicationSendDTO communicationSendDTO = new CommunicationSendDTO();
51                  communicationSendDTO.setLastEvent("EXITO");
52                  communicator.storeObject( communicationSendDTO );
53              }catch( Exception exc ){
54                  log.error("ERROR: ", exc);
55                  CommunicationSendDTO communicationSendDTO = new CommunicationSendDTO();
56                  communicationSendDTO.setLastEvent("ERROR");
57                  communicationSendDTO.setMessage( exc.getCause().toString() );
58                  communicator.storeObject( communicationSendDTO );
59              }
```

Figura 13 Invocando el main del alumno

En la línea 43, puede observarse que obtenemos, de la clase Main del programa del alumno, el método main que corresponde al punto de entrada del programa principal a inspeccionar. Luego, en la línea 48, este es invocado. El llamado al Main retorna cuando termina (Terminando el proceso de Debug) o bien cuando levanta una excepción no controlada por el alumno (Situación de error), por esto mismo, es que se encuentra ubicado dentro de un bloque *try-catch.*

Plugin

Pensando en la comodidad de los alumnos, se pensó en el desarrollo de un plugin para Eclipse de manera que pueda ser accedido sin necesidad de salir del ambiente de trabajo.

Como se explicará más adelante, se crea una vista para que pueda ser accedida sin ningún inconveniente; de esta manera, le damos la posibilidad de tener el código que ellos realicen y la vista en simultáneo, permitiéndoles flexibilidad a la hora de realizar algún cambio y post visualización del mismo.

SWT

Es un framework de código abierto para Java diseñado con el propósito de lograr un acceso eficaz y portátil para las instalaciones de interfaz de usuario de los sistemas operativos en los que se aplique. Una de sus principales características se centra en que soporta diversas plataformas tales como Linux, Windows y Mac OS.

En el capítulo 8, se abordará con más detalle este tema, explicando su uso en nuestro desarrollo.

Capítulo 6 - Plugin

6.1. Creación de un proyecto para el desarrollo de un Plug-in (Plug-in Project)

Para crear un proyecto plugin [12], se debe abrir el ambiente de trabajo (Eclipse), e ir a la opción file → new → Plug-In Project.

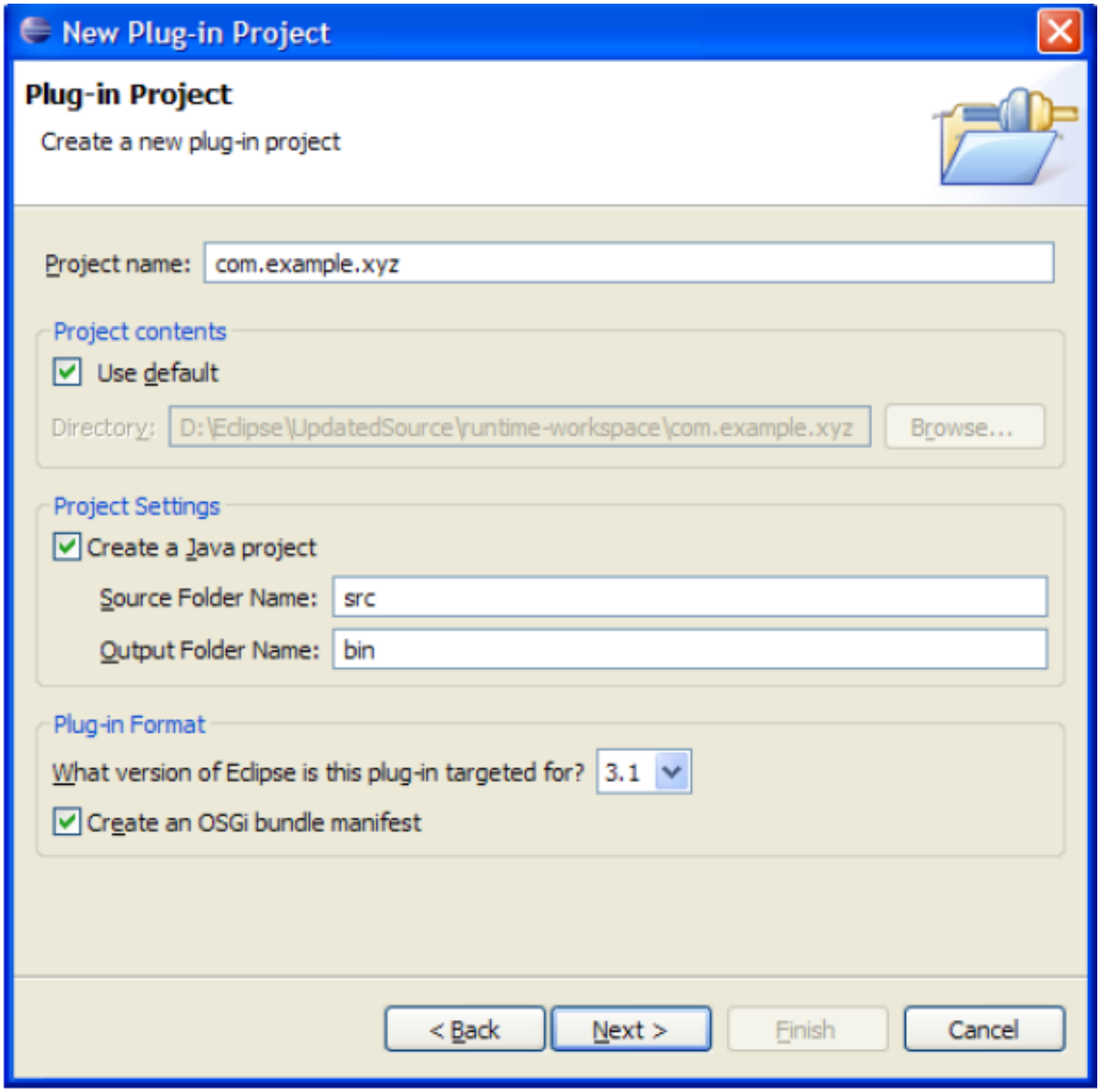

Figura 14 Nuevo proyecto Plug-in

Se tiene que establecer un nombre de proyecto para que nos dé la opción de seleccionar si deseamos crear un proyecto java o un proyecto simple. Debido a que la

mayoría de los plugins están preparados para contener código Java, se selecciona dicha opción.

Una vez completado el proceso, se presiona el botón “next” donde nos mostrará una nueva ventana de configuración.

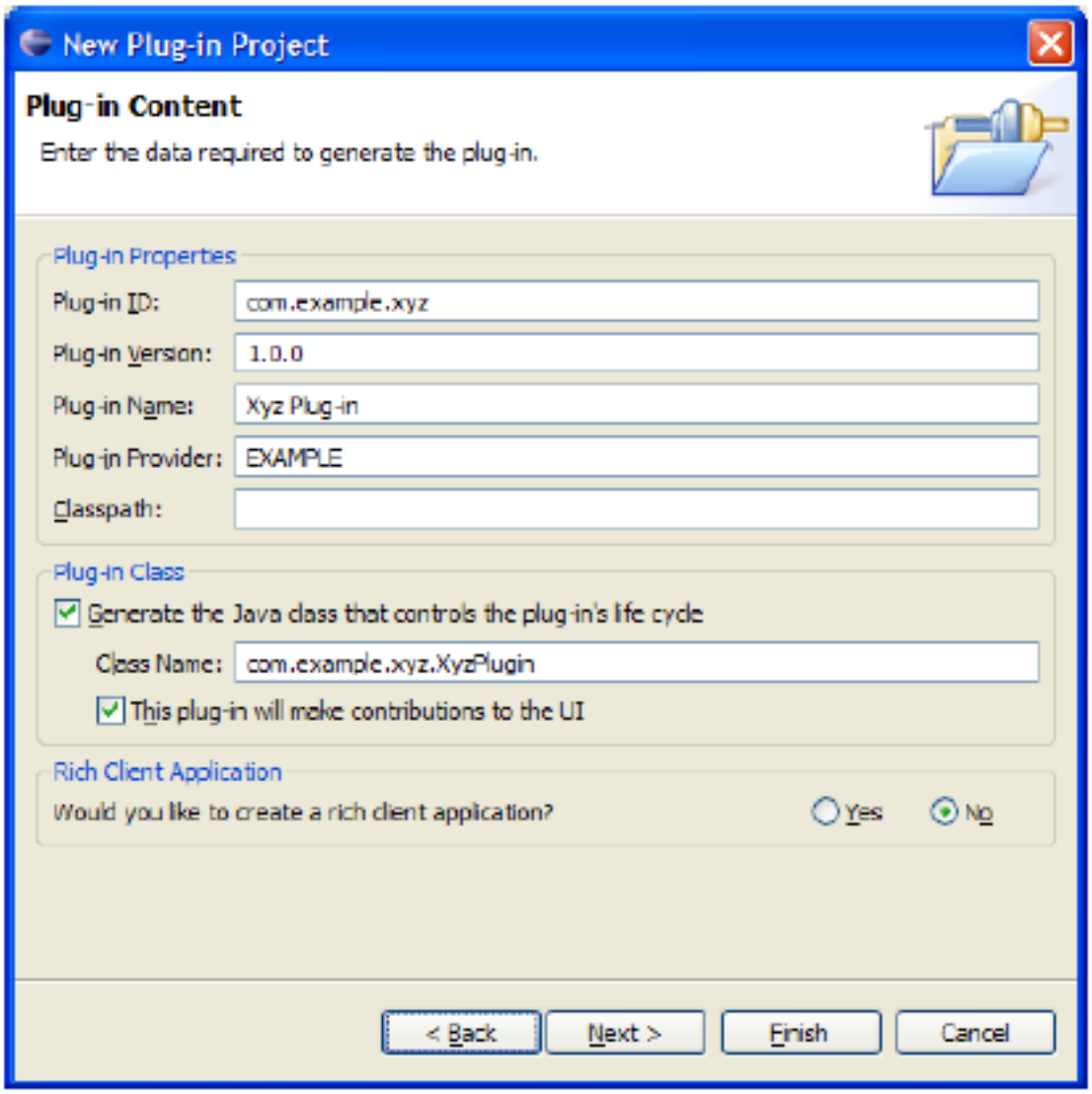

Figura 15 Ventana de configuración

Se debe establecer los campos necesarios con los cuales el archivo plugin.xml es inicializado; incluyendo el plugin-id, la versión y el nombre. Al dejar el campo classpath vacío, se crea un solo archivo jar con todas las clases y recursos en la raíz.

Nuevamente al completar los campos necesarios, se presiona en el botón “next”.

Es en este momento donde se tiene la opción de elegir alguno de los templates que nos ofrece el PDE (Plugin Development Environment).

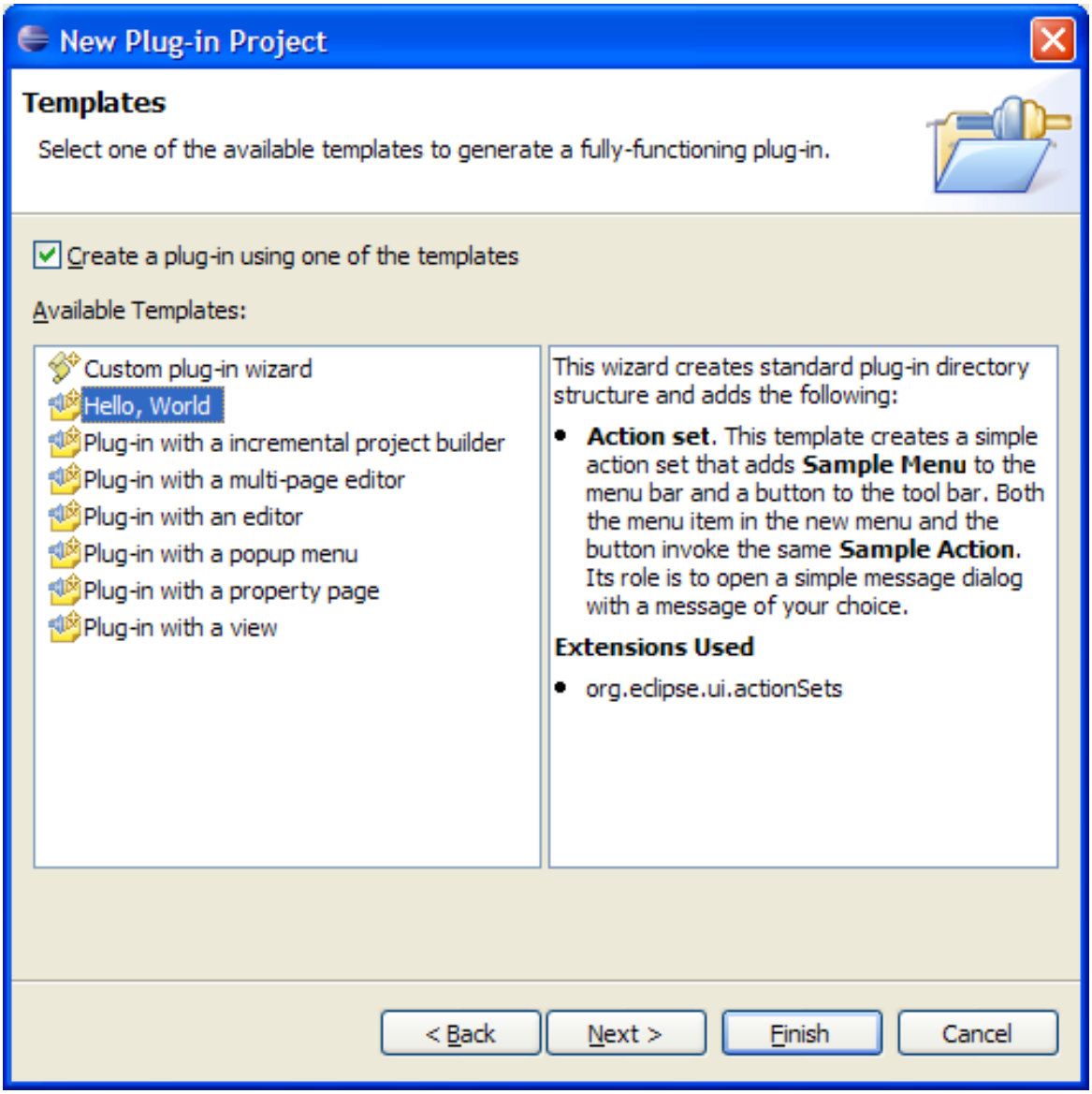

Figura 16 Template del plug-in

Estructura del Proyecto

Una vez finalizado el asistente, la estructura inicial del proyecto se asemeja al de la figura 17.

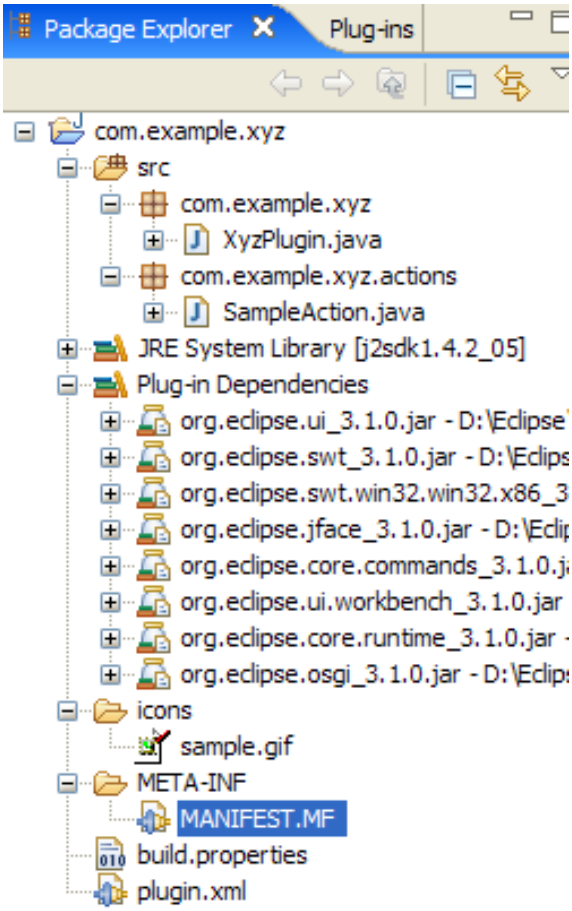

Figura 17 Estructura del proyecto

Editar MANIFEST.MF

El eclipse nos provee de un editor para el archivo MANIFEST.MF. Este editor de varias páginas es el lugar central para gestionar el plugin. Cuando se usan los campos de edición, transparentemente PDE(Plugin Development Environment) escribe los cambios en los archivos correspondientes.

- Overview

La vista “Overview”: Está diseñada para tener una rápida referencia de cómo se desarrolla, testea y despliega un plugin.

The Plugin Content: En esta sección se explica la estructura y el contenido de cada sección del manifest.

The Testing: Provee accesos directos que permiten iniciar rápidamente un ambiente de trabajo en tiempo de ejecución, para testear y “debuggear” el plugin.

The Exporting: Enumera los pasos necesarios, para construir y empaquetar el plugin.

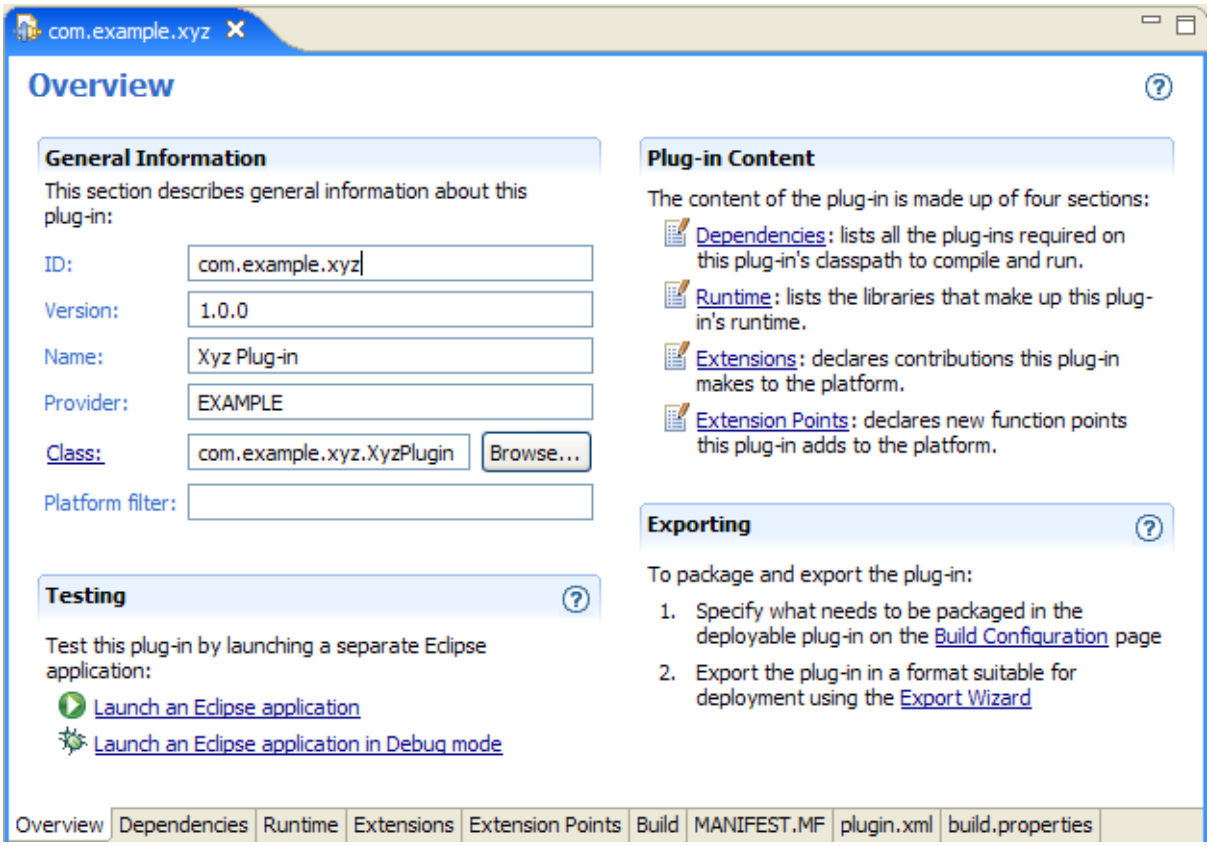

Figura 18 Overview

- Dependencies

Esta página muestra las dependencias que son necesarias para el plugin. Cuando se agrega una nueva, y se guarda el archivo, PDE(Plugin Development Environment) automáticamente actualiza el classpath.

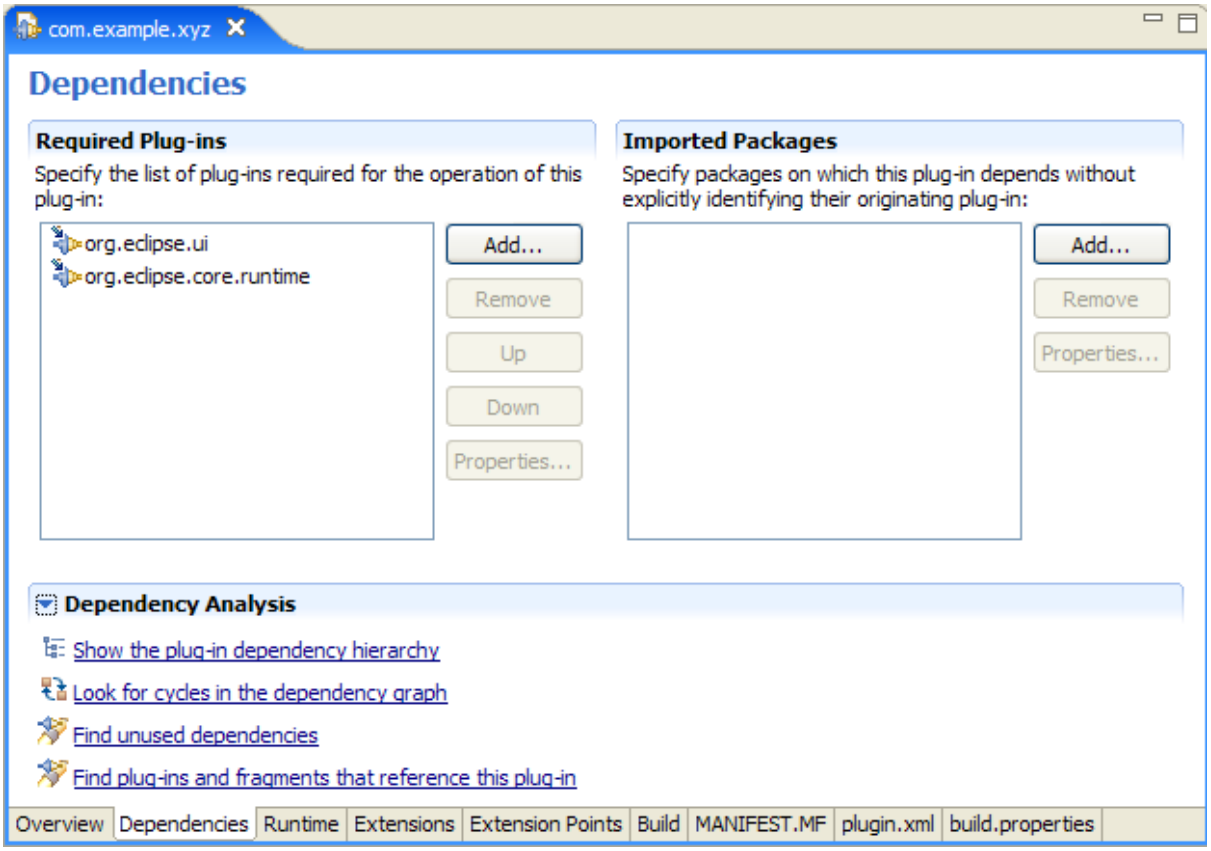

Figura 19 Dependencies

- Runtime

Se encarga de mostrar todos los paquetes que van a ser visibles por los demás plugins.

Package Visibility: Esta sección nos permite controlar, en función de cada paquete la visibilidad del código de nuestro plugin.

Classpath Section: Es el lugar donde se declaran los nombres de las librerías que constituyen el classpath del plugin.

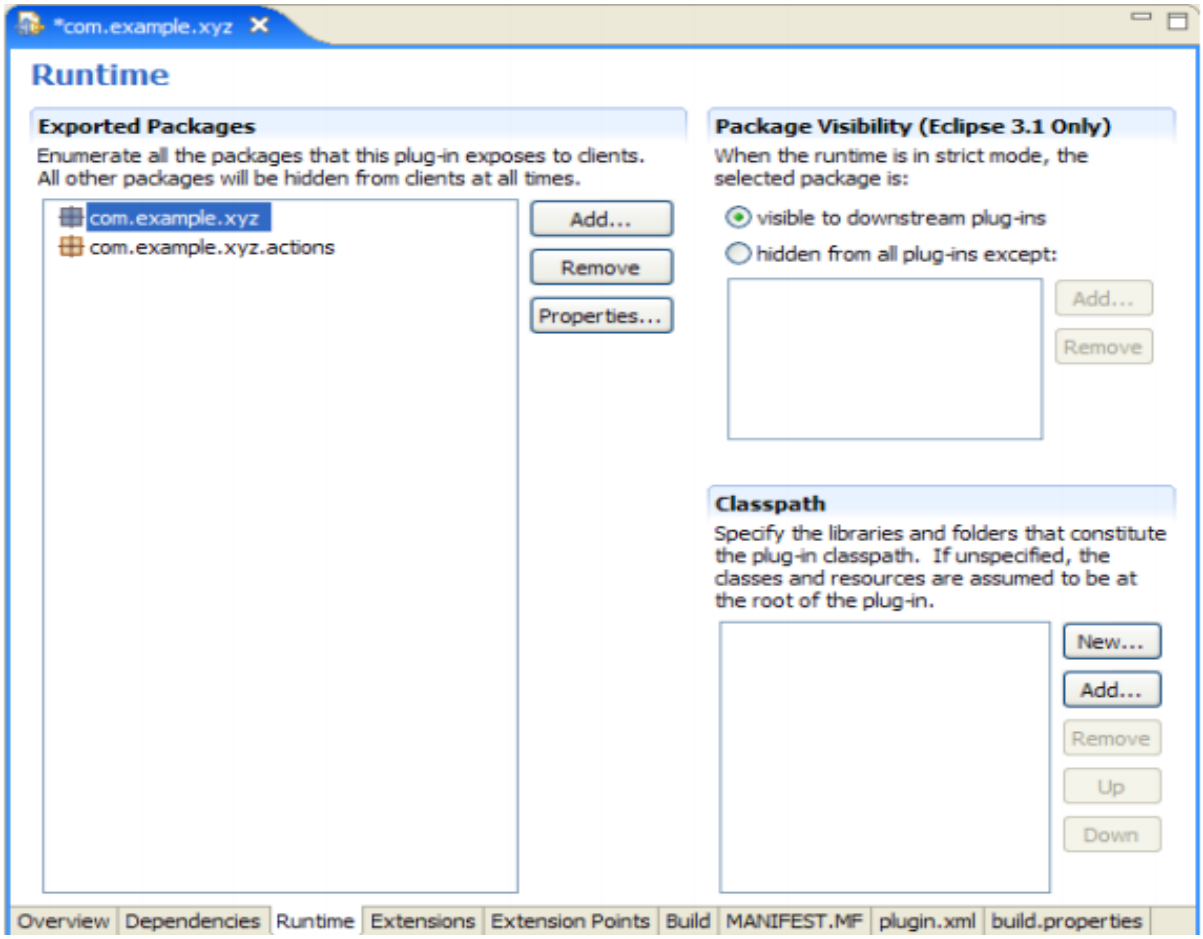

Figura 20 Runtime

Restricción de acceso

La versión 3.1 de Eclipse da la posibilidad de controlar en función de cada paquete, la visibilidad del código del plugin en relación con los que se encuentran por debajo.

Un paquete se puede clasificar como una de las siguientes maneras:

- Accesible.
- Prohibido.
- Interno
- Interno con amigos.

Accessible Packages

Son visibles para aquellos paquetes que se encuentren por debajo del plugin que se está desarrollando. Mientras que los paquetes de la API deben caer en esta categoría, es responsabilidad del desarrollador, decidir que otros paquetes exportados por el plugin serán accesibles.

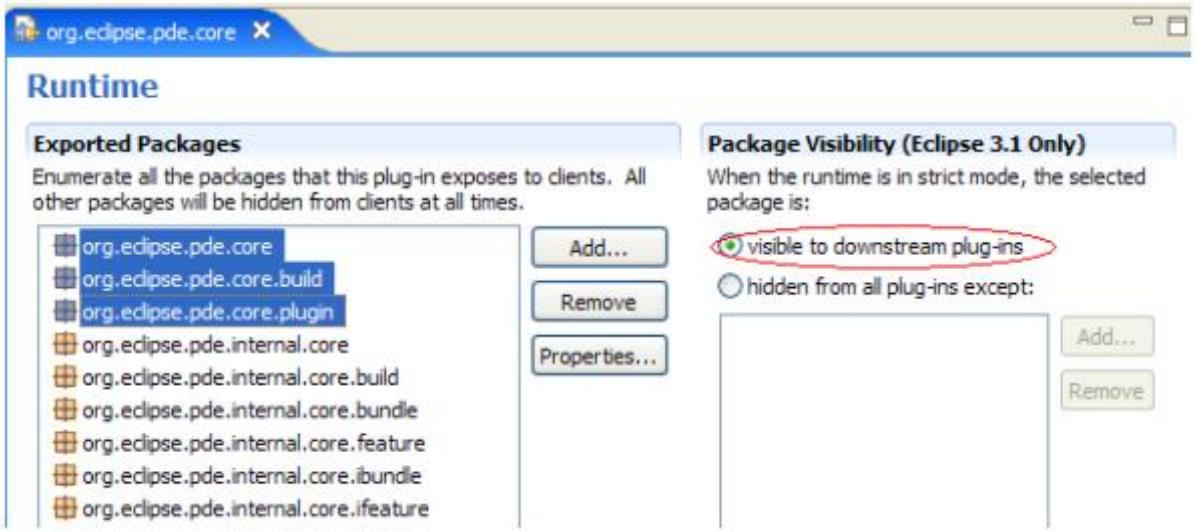

Figura 21 Paquetes accesibles

Forbidden package

Se puede ocultar un paquete de los plugins subyacentes, excluyéndolos de la lista en la sección de paquetes exportados.

Cuando se realiza una referencia a un tipo de un paquete que se encuentra oculto, se produce un error en tiempo de ejecución.

Paquetes Internos

Son paquetes que no están destinados para el uso de plugins subyacentes.

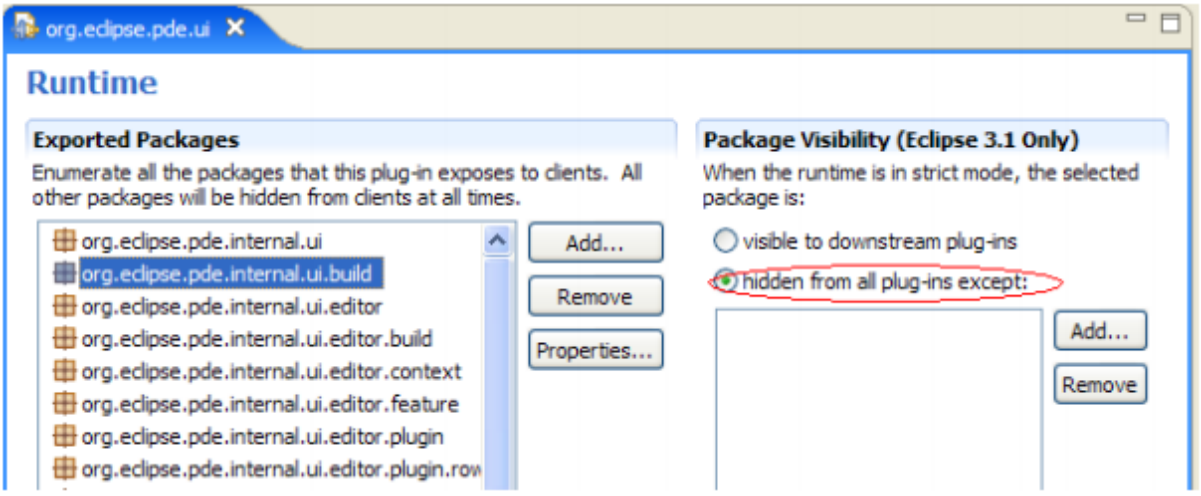

Figura 22 Paquetes internos

• Extensions page

Las extensiones son el mecanismo central que aportan al comportamiento de la plataforma. Cada vez que se quiera agregar nuevo comportamiento al plugin, el mismo deberá ser establecido como una extensión.

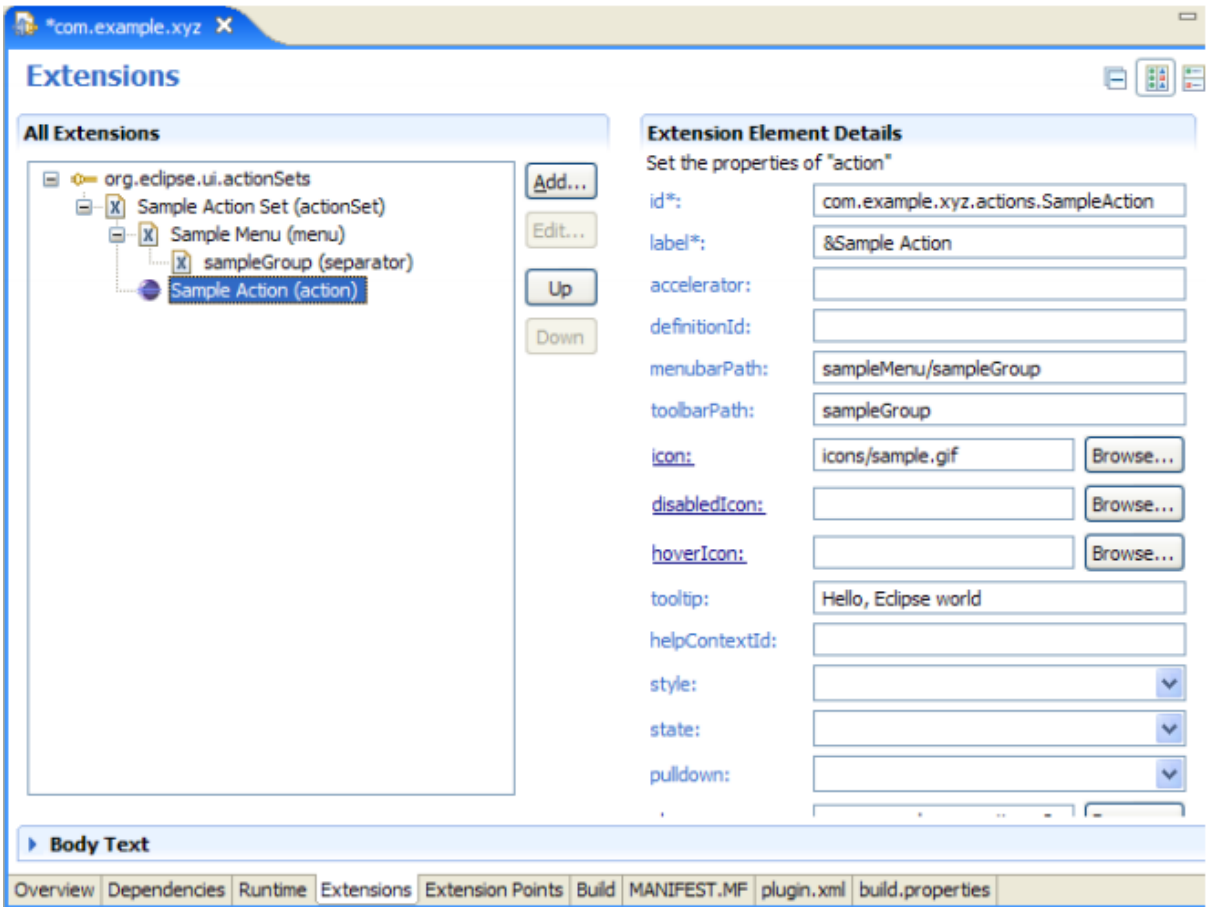

Figura 23 Extension points

• Extension point page

Los puntos de extensión definen nuevos puntos de funcionalidad de manera que otros plugins puedan conectarse. A su vez, permiten agregar nuevas funcionalidades al plugin.

Posee tres atributos:

1. ID: Atributo requerido donde el valor es un simple nombre.

2. NAME: Atributo requerido cuyo valor es un string traducido.

3. SCHEMA: Atributo opcional cuyo valor es un path relativo al esquema correspondiente a este punto de extensión. Aunque sea un atributo opcional es recomendable establecerlo, de manera que PDE (Plugin Development Environment) pueda usarlo y así sea más fácil para los desarrolladores usar un punto de extensión.

Extension Point "org.eclipse.ui.console.console view"

El paquete nos provee de un conjunto de clases e interfaces que facilitan la creación de las consolas en la "ConsoleView". Contiene una consola abstracta que nos brinda de funcionalidad básica, como también dos completas implementaciones de consolas que pueden ser subclasificadas.

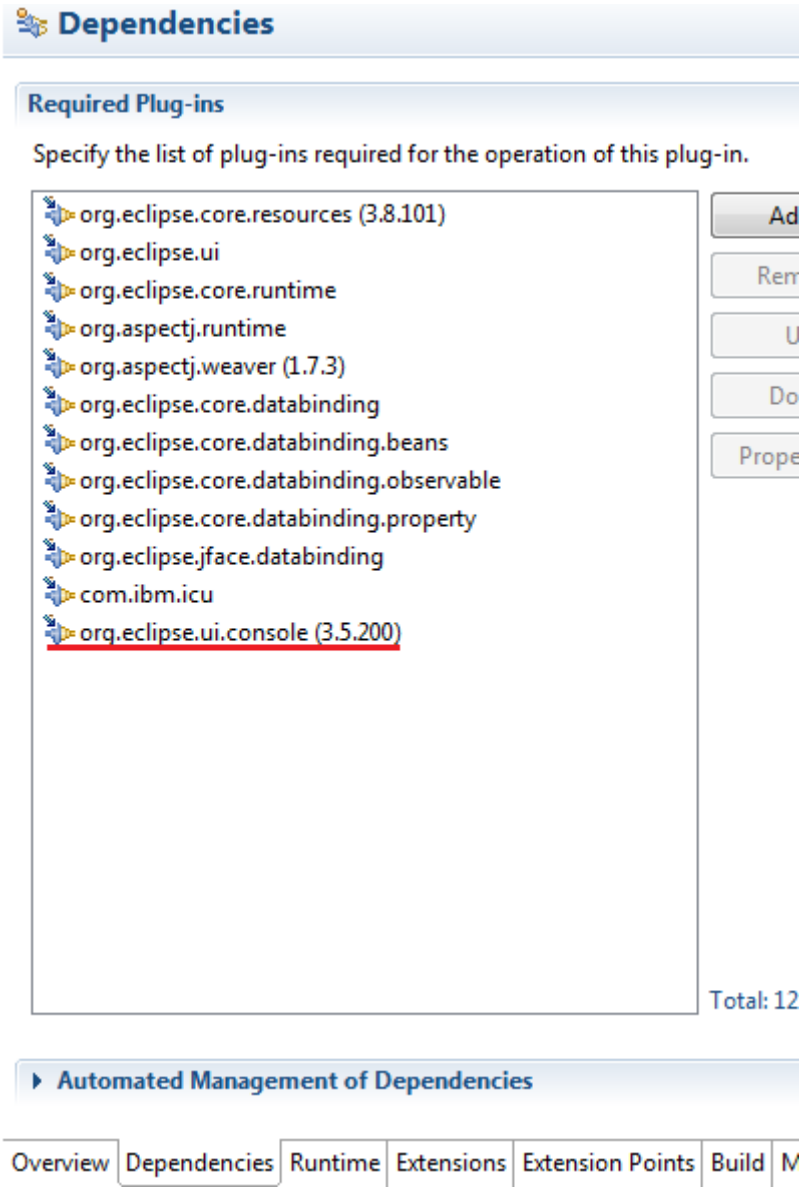

Figura 24 Dependencias de Algolipse

En la figura 24, puede verse que se agrega la dependencia del plugin hacia el extension points que permite integrarse con consolas en Eclipse, con el objetivo de imprimir en la misma el "output" del código del alumno; mientras que en la figura 25, se encuentra el código que fue realizado para poder escribir en la consola.

```
279         if (arg1.toString().equals("run")) {
280             /*Se borra la lista de items cada vez que se presiona sobre run*/
281             Display.getDefault().asyncExec(new Runnable() {
282                 public void run() {
283                     for (ExpandItem elements : expandBar.getItems()){
284                         elements.getControl().dispose();
285                         elements.dispose();
286                     }
287                     instanceNumber=0;
288 
289                     //Borra el panel de ultmima instancia
290                     for (Control elements : panelUltimaInstancia.getChildren()){
291                         elements.dispose();
292                     }
293 
294                     //Preparamos para observar el console Thread
295                     IConsoleManager conMan = ConsolePlugin.getDefault().getConsoleManager();
296                     MessageConsole myConsole = new MessageConsole("Run", null);
297                     conMan.addConsoles(new IConsole[]{myConsole});
298                     myConsoleStream = myConsole.newMessageStream();
299                     GeneralView.this.setConsoleIterator( model.getExecuteManager().getConsoleIterator() );
300                 }
301             });
302         }
303 
304         if (arg1.toString().equals("consoleChanged")) {
305             Display.getDefault().asyncExec(new Runnable() {
306                 public void run() {
307                     myConsoleStream.print( GeneralView.this.getConsoleIterator().next() );
308                 }
309             });
310         }
```

Figura 25 Uso de la extension Console

Este fragmento de código se encuentra dentro del observador que responde al patrón Observer. En la primera sentencia if, maneja el evento que indica el inicio de la ejecución, y pide la creación de una nueva instancia de Console (Linea 295-298). En la línea 299, se construye un iterador para recorrer la lista de string de la misma. En la segunda sentencia if, se maneja el evento que indica el cambio, por lo que se le pide al iterador el siguiente elemento.

Build configuration page

Como se muestra en la figura 26, el build configuration page contiene toda la información necesaria para construir los paquetes y exportar el plugin.

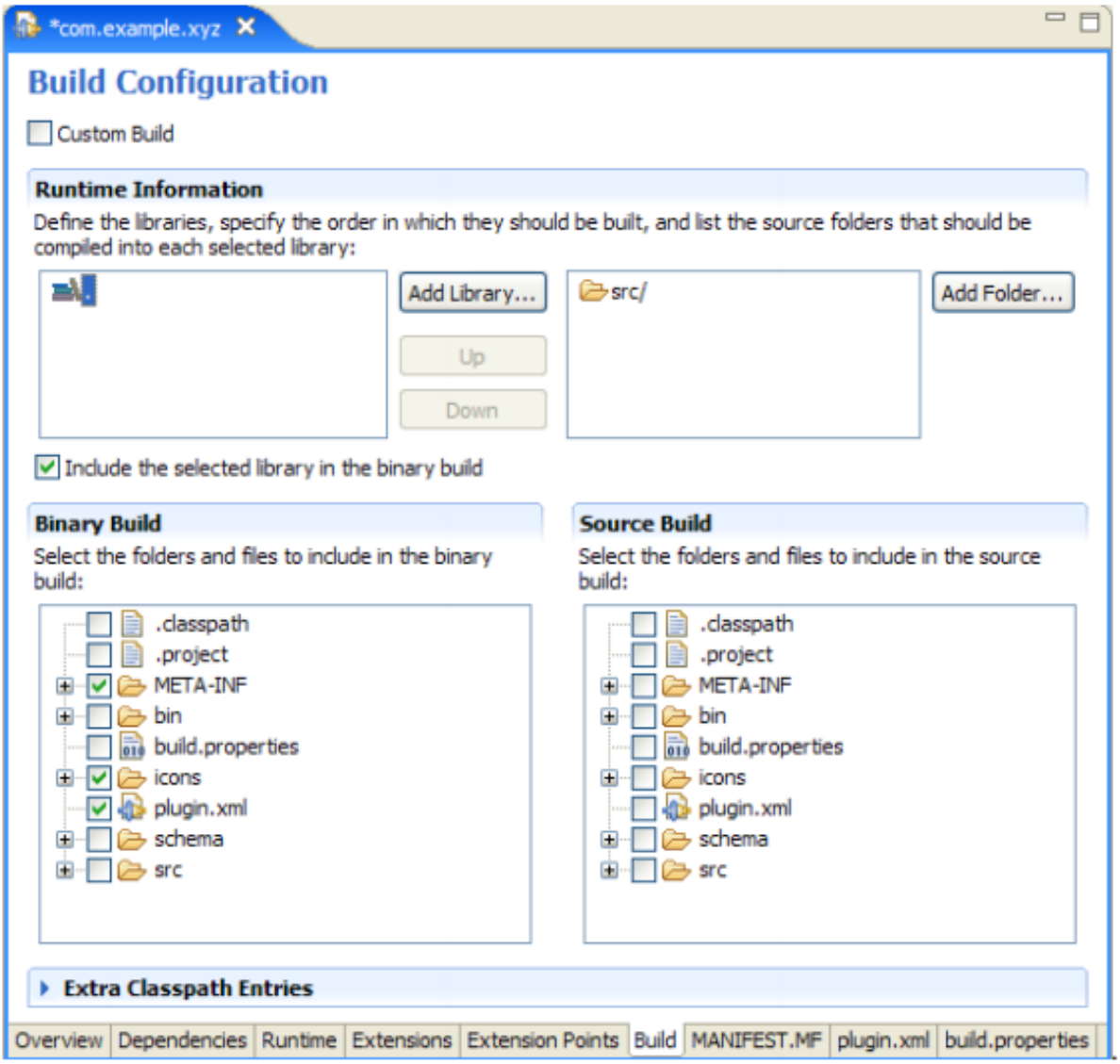

Figura 26 Build Configuration Page

La sección Runetime information, lista todas las librerías que se necesitan para realizar el "build". Para cada una de ellas, se debe especificar la carpeta "source" que será compilada.

La sección "Binary Build", se van a listar todos los archivos y carpetas que serán empaquetadas dentro del plugin.

MANIFEST.MF

Es aquí donde se almacenan todos los datos del plugin y sus dependencias.

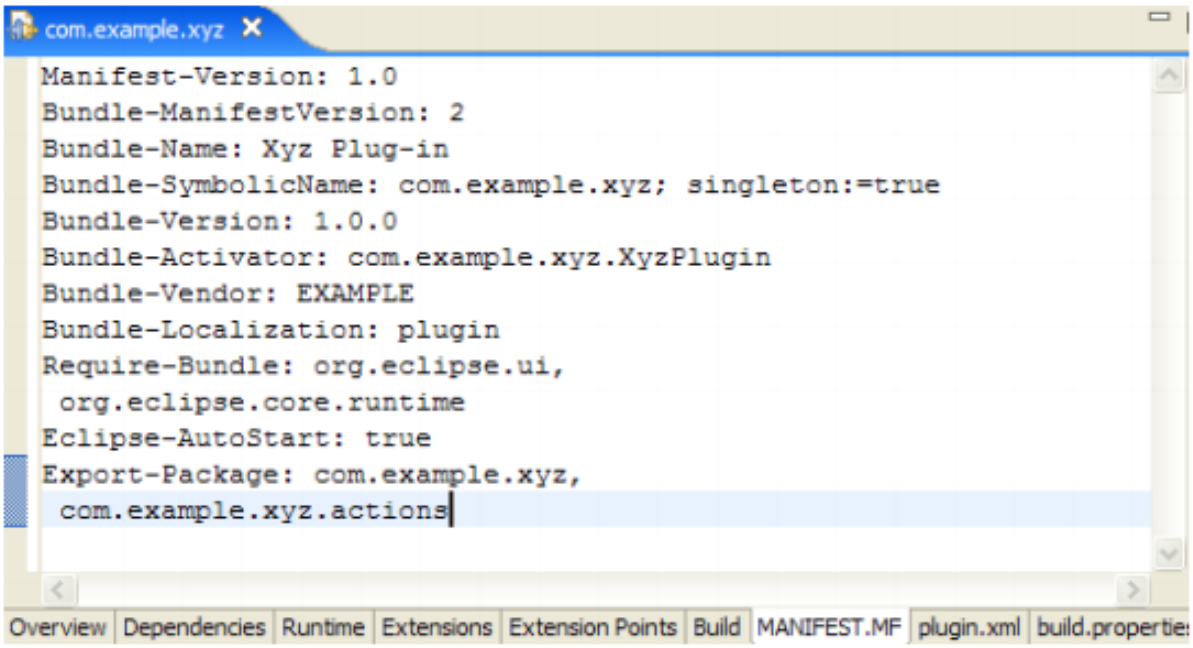

Figura 27 Manifest

Plugin.XML

En la figura 28, vemos un ejemplo del archivo plugin.xml, el cual se encarga de contener los puntos de extensión y las extensiones de plugin.

com.example.xyz

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>

   <extension-point id="parsers" name="Parsers" schema="schema/parsers.exsd"/>

   <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Sample Action Set"
            visible="true"
            id="com.example.xyz.actionSet">
         <menu
               label="Sample &amp;amp;Menu"
               id="sampleMenu">
            <separator
                  name="sampleGroup">
            </separator>
         </menu>
         <action
               label="&amp;amp;Sample Action"
               icon="icons/sample.gif"
               class="com.example.xyz.actions.SampleAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               toolbarPath="sampleGroup"
               id="com.example.xyz.actions.SampleAction">
         </action>
      </actionSet>
   </extension>

</plugin>
```

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | build.properties

Figura 28 Plugin.xml

Build.properties

Contiene toda la información necesaria para construir el plugin.

com.example.xyz

```
source.. = src/
output.. = bin/
bin.includes = plugin.xml,\
               META-INF/,\
               .,\
               icons/
```

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | plugin.xml | build.properties

Figura 29 Build properties

6.2. Generar vistas y perspectivas en Eclipse

Las vistas son típicamente utilizadas para navegar una jerarquía de información, abrir un editor, o monitorear propiedades del editor activo. Para el desarrollo en cuestión, la misma será utilizada para contener el plugin anteriormente desarrollado.

El entorno de trabajo contiene ciertos componentes, los cuales sirven para demostrar el rol de una vista. Por ejemplo, la vista “navigator” es utilizada para realizar monitoreos y navegaciones a través del “workspace”. Al seleccionar un archivo en esta vista, se puede navegar la estructura utilizando la vista “outline”, la cual se abrirá automáticamente.

Perspectivas en Eclipse

Una perspectiva es un contenedor visual de vistas y editores. Es como una página dentro de un libro; es decir, existen dentro de una ventana junto a otras perspectivas y solo una puede ser visible en cualquier momento.

A nivel de implementación se vuelve más complejo. La interfaz de usuario está expuesta a través de una serie de interfaces en org.eclipse.ui. La raíz de la interfaz de usuario es accedida invocando “PlataformUi.getWorkbench()”, retornando un objeto de tipo “IWorkbench”. Un “workbench” tiene una o más ventanas de tipo “IWorkbenchWindow”, donde cada una puede tener una colección de páginas del tipo “IWorkbenchPage”. Desde el punto de vista de la interfaz de usuario, una página es conocida como una perspectiva.

La estructura del “workbench” se puede ver en la figura 30.

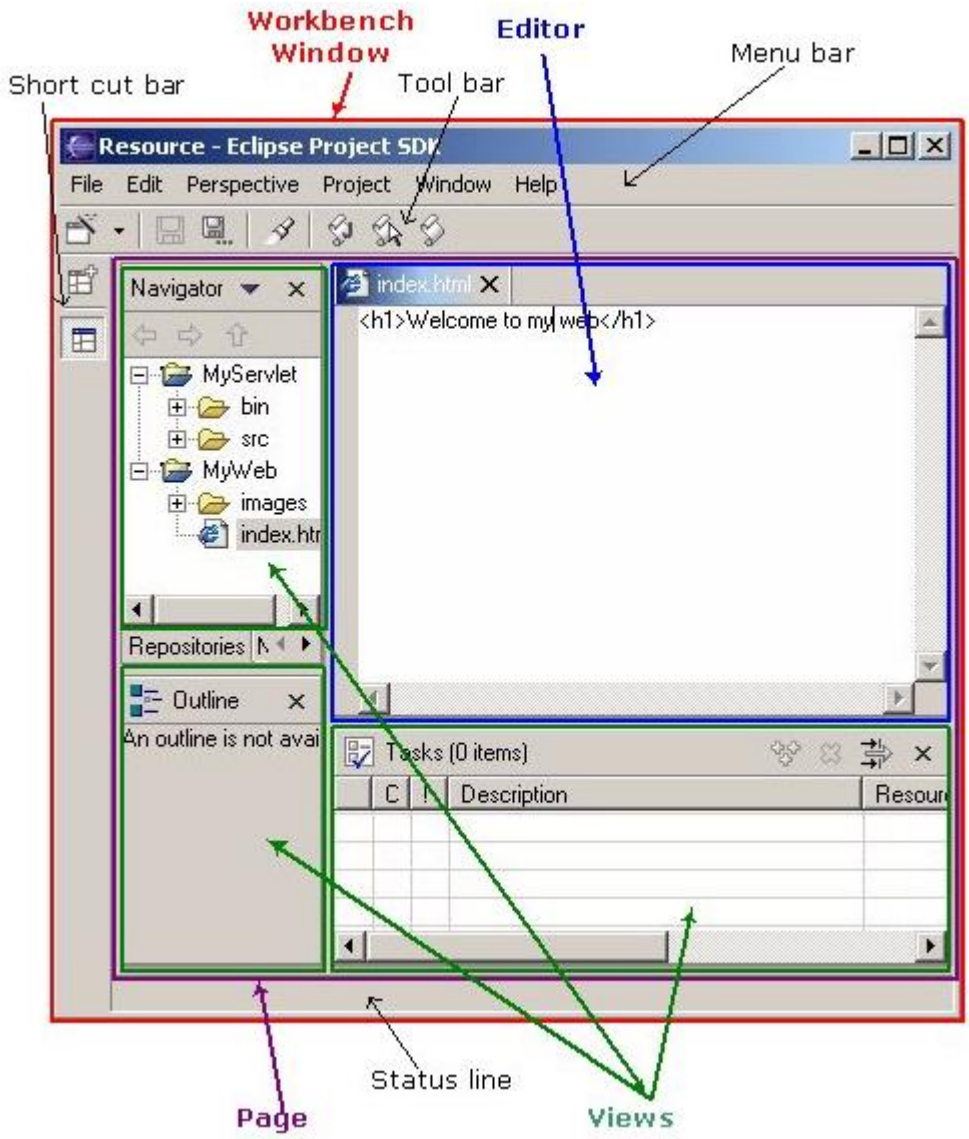

Figura 30 Estructura de UI de eclipse

La “workbench window” se encuentra marcada con el color rojo. Dentro de esta ventana hay una única perspectiva abierta, marcada con el color violeta (perspectiva Java). Mientras el “workbench” está corriendo nuevas páginas y ventanas pueden ser creadas seleccionando la opción “Open perspective” del menú principal. A su vez una página puede ser creada programáticamente usando la API pública del “IWorkbenchWindow”.

Dentro del “workbench” una nueva perspectiva puede ser definida de dos maneras. Un usuario puede agregar una nueva perspectiva (o sobrescribir una existente utilizando la interfaz de usuario de Eclipse) o agregarla al momento de desarrollar el Plug-in.

La creación de una nueva perspectiva es un proceso de tres pasos:

1. Crear un Plug-in.

2. Agregar una “extension perspective” al plugin.xml.

3. Definir una “perspective class”.

Si se desea que la nueva vista se encuentre visible, una opción es copiar una perspectiva existente. Si se toma este enfoque, el usuario no tardará en verse abrumado por las diversas opciones que trae la perspectiva, por lo que es probable, que sea ignorada y finalice creándola el mismo.

En general una perspectiva sólo debería ser creada cuando hay un cierto grupo de tareas relacionadas, las cuales serían beneficiadas de una configuración predefinida de acciones y vistas. Por ejemplo, la “Java perspective” es un buen contexto para la realización de tareas Java.

En algunas situaciones puede ser mejor usar una perspectiva existente. Por ejemplo, si se crea una única vista que pertenece al contexto de java, es probable que sea mejor agregarla a la “standard java perspective” en lugar de crear una nueva. Para nuestro desarrollo, no fue necesario crearla, debido a que sólo posee una única vista y la misma está vinculada al contexto de java. En el siguiente fragmento del plugin.xml, puede verse cómo se vincula la vista “Algolipse” con la perspectiva “JavaPerspective”

```
<extension
      point="org.eclipse.ui.perspectiveExtensions">
   <perspectiveExtension
         targetID="org.eclipse.jdt.ui.JavaPerspective">
      <view
            ratio="0.5"
            relative="org.eclipse.ui.views.ProblemView"
            relationship="right"
            id="com.fl.algolipse.plugin.views.Algolipse">
      </view>
   </perspectiveExtension>
</extension>
```

Agregar una nueva vista

Para agregar una nueva vista al “Workbench” se deben seguir estos tres simples pasos:

1. Como se explicó anteriormente, se debe crear un proyecto plugin y configurar el archivo plugin.xml.

2. En el archivo plugin.xml utilizaremos el punto de extensión “org.eclipse.ui.views” para agregar una nueva vista denominada “Label”. Este fragmento de código será agregado al archivo “plugin.xml” y contiene los atributos básicos para la vista: id, name, icon and class.

Si analizamos el código, vemos que se está creando una nueva vista con el nombre Label View a través de una extensión point.

```
<extension point="org.eclipse.ui.views">
        <view id="org.eclipse.ui.articles.views.labelview"
        name="Label View"
        class="org.eclipse.ui.articles.views.LabelView"
        icon="icons\view.gif"/>
</extension>
```

- id: Nombre único que será utilizado para identificar la vista.
- name: Un nombre traducible que será utilizado en la interfaz de usuario para la vista.
- class: Un nombre completo de la clase que implementa "org.eclipse.ui.IViewPart". Una práctica común es subclasificarlo en "org.eclipse.ui.part.ViewPart ".
- icon: Nombre relativo del icono que será asociado a la vista.

3. Para nuestra ayuda, la plataforma contiene una clase abstracta denominada "org.eclipse.ui.part.ViewPart", la cual implementa la mayoría del comportamiento por defecto. A su vez, nos brinda de un método "createPartControl" el cual crea un objeto SWT Label donde mostrará el mensaje "Hello World".

Una vez que la clase ha sido declarada y compilada, se podrán testear los resultados. Para esto se debe invocar a la perspectiva windows → show view→ other--> Label View. De esta manera la vista se instancia y abre en la perspectiva actual.

6.3. Creación de un proyecto para empaquetar a un plugin (Feature Project)

A la hora de desplegar el plugin, crearemos un "Feature Project". Es a través de este dónde podremos realizar actualizaciones y adiciones a nuestro plugin.

Similar a la creación de un proyecto plugin, PDE (Plugin Development Environment) trata a los "features" como proyectos. Para realizar la creación:

1. New−>Project−>Plug−in Development−>Feature Project.
2. Agregar un nombre de proyecto y presionar "siguiente" (com.example.feature).

3. Agregar un nombre de “feature” y su versión.

4. En la siguiente página, chequear el plugin y el fragmento.

5. Presionar “finish”.

A partir de este momento, se genera el proyecto feature en el workspace.

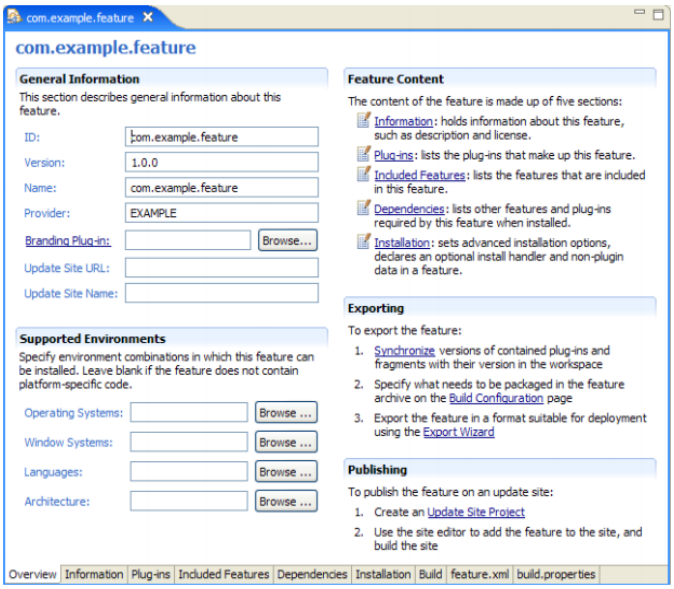

Figura 31 Feature manifest editor

La información agregada cuando se crea el proyecto, puede ser configurada desde la vista “Overview”. A su vez se pueden agregar nuevas características como el “Update Site URL”, para que sea usado por el Update Manager cuando se buscan nuevas actualizaciones.

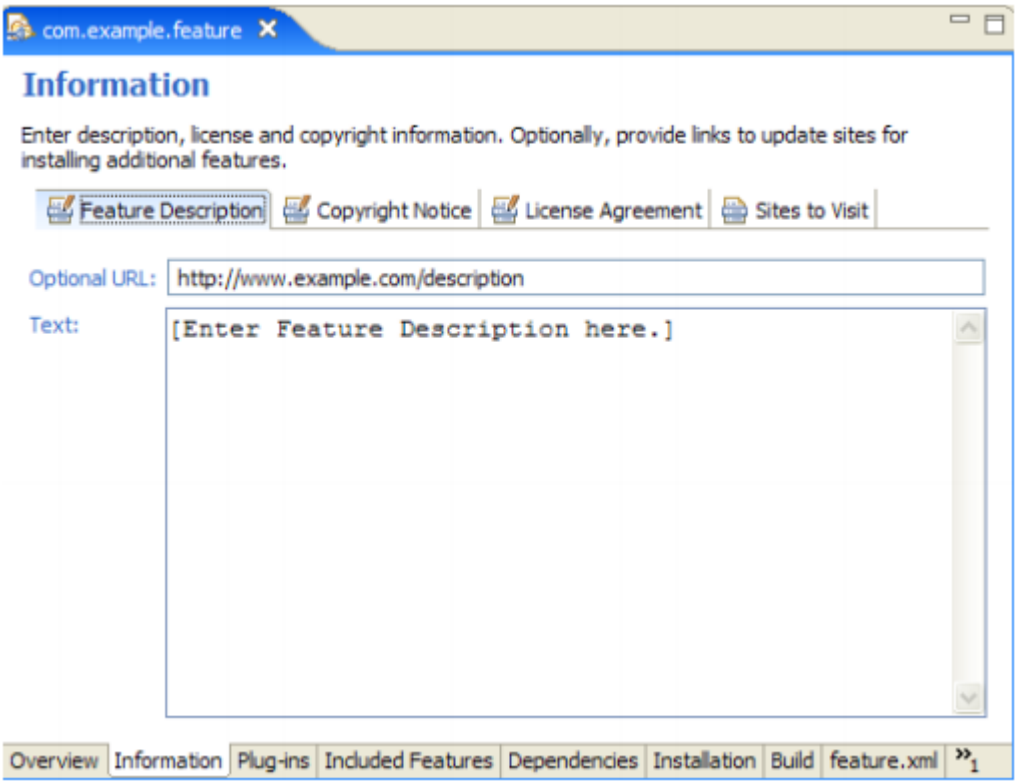

Figura 32 Página de información del Feature

En esta sección podremos agregar información de licencia, derechos de autor, como también una descripción del "feature" que se está creando. Cada una de estas categorías, puede ser representada como un texto, o como una URL donde aloje una página HTML válida.

6.4. Prueba de un Plug-in

Cuando el Plug-in está listo para ser testeado, se puede lanzar una instancia de Eclipse separada que contiene el Plug-in instalado. Para esto, se debe abrir la ventana "run configuration", para luego presionar sobre el botón "run".

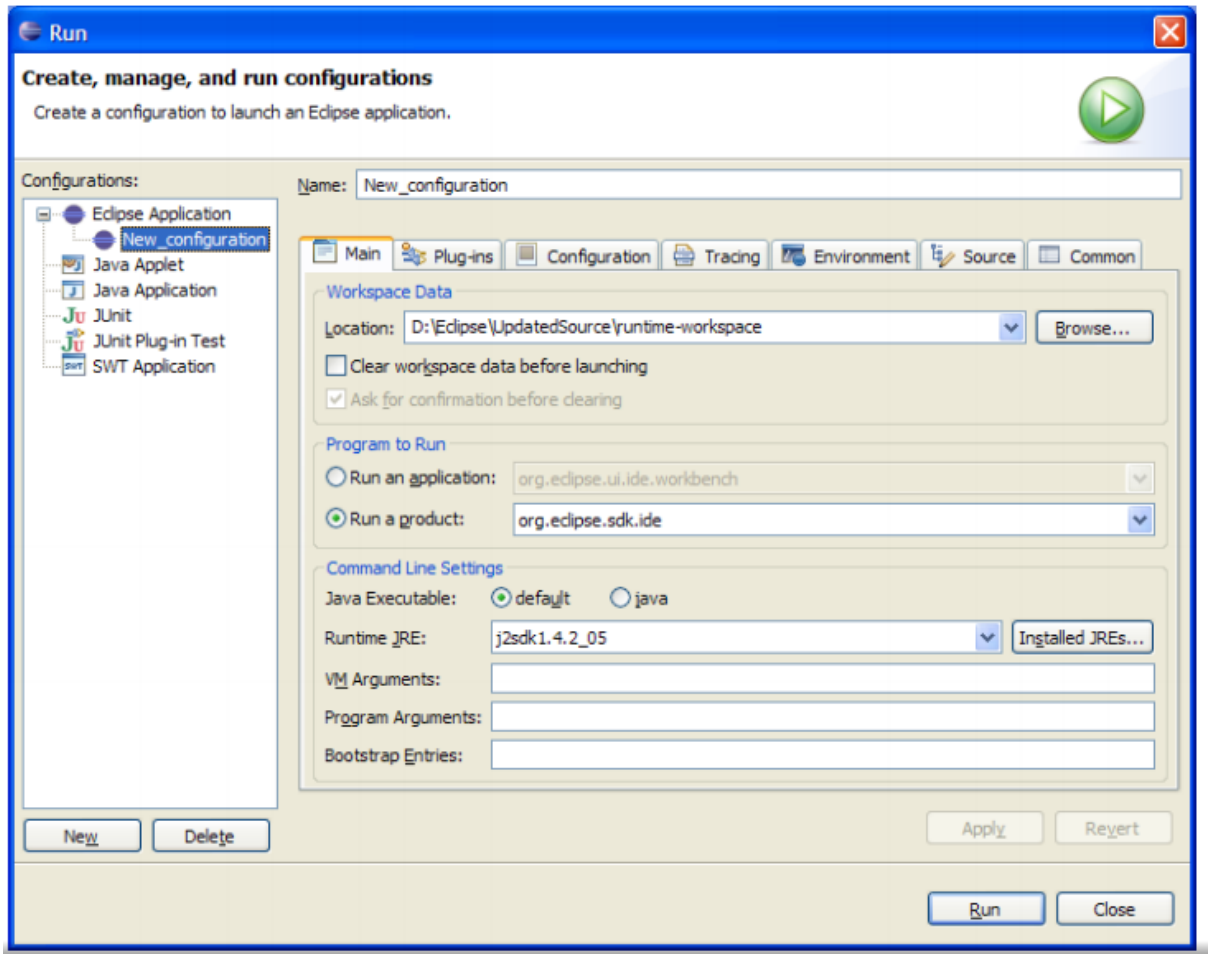

Figura 33 Run configuration del proyecto de plugin

6.5. Creación de un proyecto para desplegar un plugin (Update Site Project)

Eclipse es capaz de instalar o actualizar “features” ubicados en servidores remotos (o locales). Los “features” y plugins deben ser empaquetados como archivos jars y tener un manifest (site.xml) que los una entre sí. Estos archivos forman un “Update site”.

PDE (Plugin Development Environment) provee soporte para construirlos directamente en el workspace. Normalmente, estos son situados sobre servidores remotos HTTP, como así también pueden ser ubicados en un directorio local y ser vistos por el “update manager”.

Por defecto, es creado localmente en el workspace. Es posible que se quiera manejar múltiples versiones de plugins y features en este sitio, para lo cual se puede establecer el “update site” fuera del mismo.

Los pasos necesarios para la creación de un proyecto de este tipo son:

1. Abrir el “Update site” wizard vía “new → Project → Plug-in Development → Update Site Project”

2. Establecer un nombre al proyecto.

3. Establecer una ubicación.

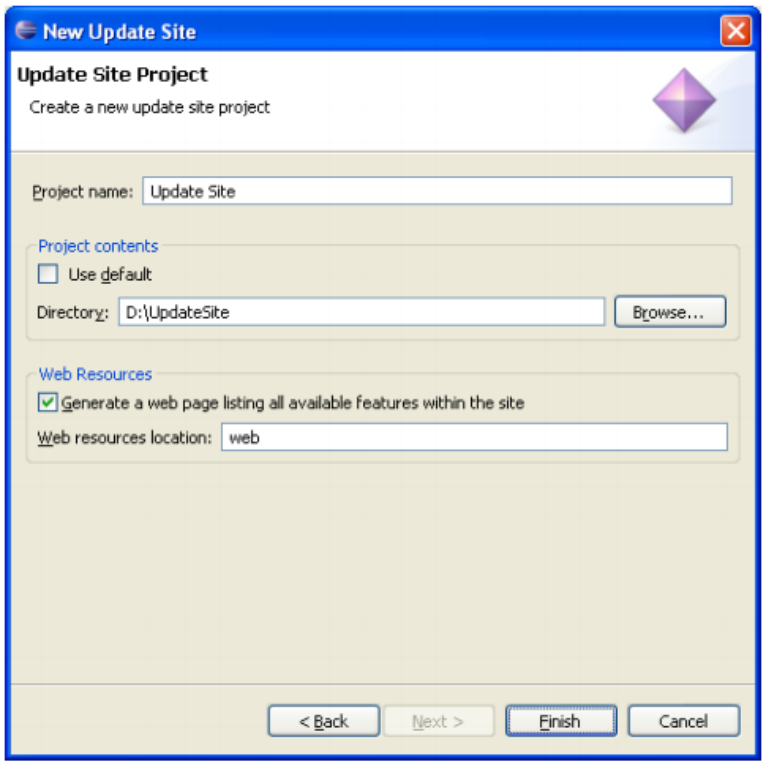

Figura 34 Creando proyecto UpdateSite

Finalmente, compilar y construir el plugin, es una tarea relativamente simple y la mayor parte del trabajo se realiza a través de la siguiente ventana.

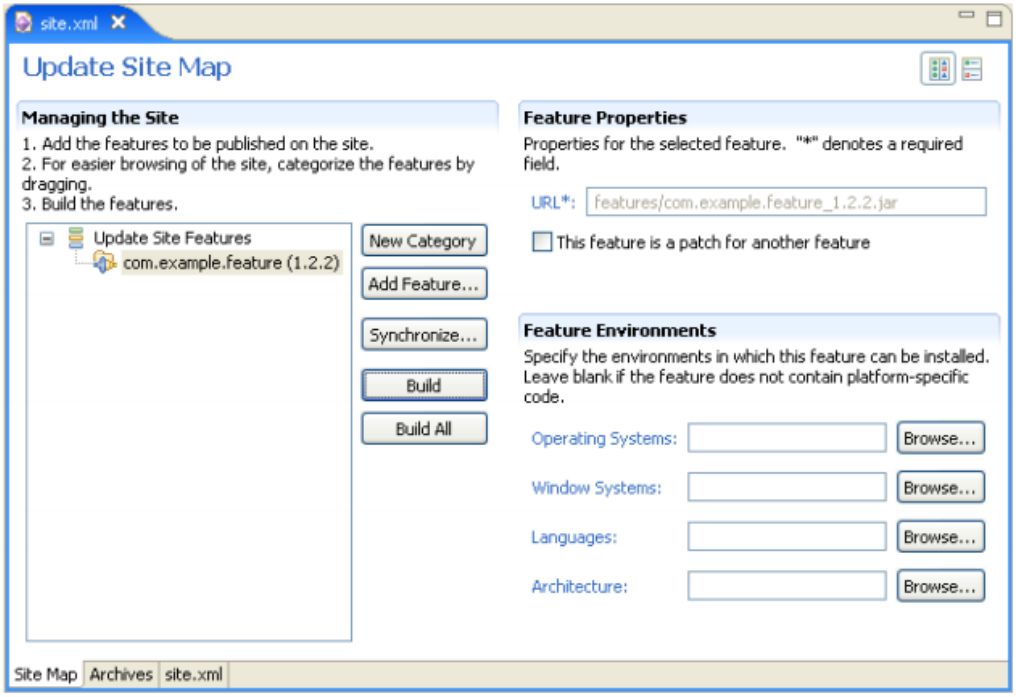

Figura 35 Configuración UpdateSite

Los “features” agregados a la sección de administración del sitio (Site.xml) serán construidos recursivamente cuando se presione “Build all”. Esto significa que el “feature” y todos los plugins que este incluye, serán construidos en una operación batch. Los jars del feature serán ubicados en el directorio “feature/” del “site project” y los jars del plug-ins serán ubicados en el directorio “plugins/”.

Para una búsqueda más sencilla del feature en el “update site”, se pueden crear categorías de manera que sirvan para organizar los features a través de las mismas.

6.6. Puesta en Marcha del Plugin

Para realizar la puesta en marcha del plugin, el alumno deberá tener una versión de Eclipse instalada para luego poder realizar una serie de pasos que lo lleven a instalar el mismo.

Pasos para la instalación.

1. Desde la ventana install new software de Eclipse (Help → install new Software):

   a. Agregar la URL local referenciando el directorio que contiene el Update Site del plugin (Site.xml).

b. El plugin depende del lenguaje de programación AspectJ, pero normalmente no se encuentra en los repositorios oficiales de Eclipse, por lo que también se debe agregar la URL para descargarlo.

c. Seleccionamos el repositorio local y aparecerá el plugin bajo la categoría AyED. Instalamos el mismo, y este resolverá las dependencias por sí mismo.

2. Reiniciar el Eclipse y abrir la view “Algolipse”.

3. Copiar el archivo Interprete.jar en el home del usuario (Normalmente C:/Users/<nombre_del_usuario>).

4. Copiar el archivo AspectAlgolipse.jar en el home del usuario (Normalmente C:/Users/<nombre_del_usuario>).

Capítulo 7 - Arquitectura

7.1. Vista de alto nivel

La herramienta ha sido implementada en Java y diseñada de manera modular, de forma que la introducción de nuevas estructuras de datos sea lo más sencilla posible para el desarrollador. En una primera versión contendrá a los Árboles Binarios, Arboles Binario de Búsqueda, Arboles Generales y Listas, pero fue pensada para que la misma pueda crecer e incluir nuevas estructuras como Grafos.

El alumno se comunica con la herramienta a través de una vista de Eclipse, la cual consta de una ventana interactiva donde se realizará la configuración, un panel de comandos básicos y un visualizador de estructuras. Para realizar la interfaz, se utilizaron componentes *SWT* (Ver capítulo 8).

Una vez finalizado el proceso de configuración, la misma queda guardada en el modelo, que será utilizada posteriormente para realizar la ejecución. El alumno envía órdenes a través del panel de ejecución, donde serán recibidas por un módulo encargado de la ejecución de dichas órdenes. A su vez existe otro módulo (extensible según la estructura) que se encarga de procesar la estructura de datos a ser analizada.

Quien realiza la ejecución propiamente dicha, es un proceso Java independiente al creado por Eclipse (una nueva Java Virtual Machine), el cual a través de Reflection interpreta el código del alumno y con el uso de AspectJ detecta que el método seleccionado fue alcanzado, deteniéndose e informando a través de un Socket al plugin.

Para evitar que el proceso de Eclipse se bloquee en el Socket a la espera de comunicaciones, se dispara un nuevo thread encargado de realizar dicha tarea.

La comunicación entre ambos procesos se realiza serializando objetos Java que se envían a través Sockets.

7.2. Modelos

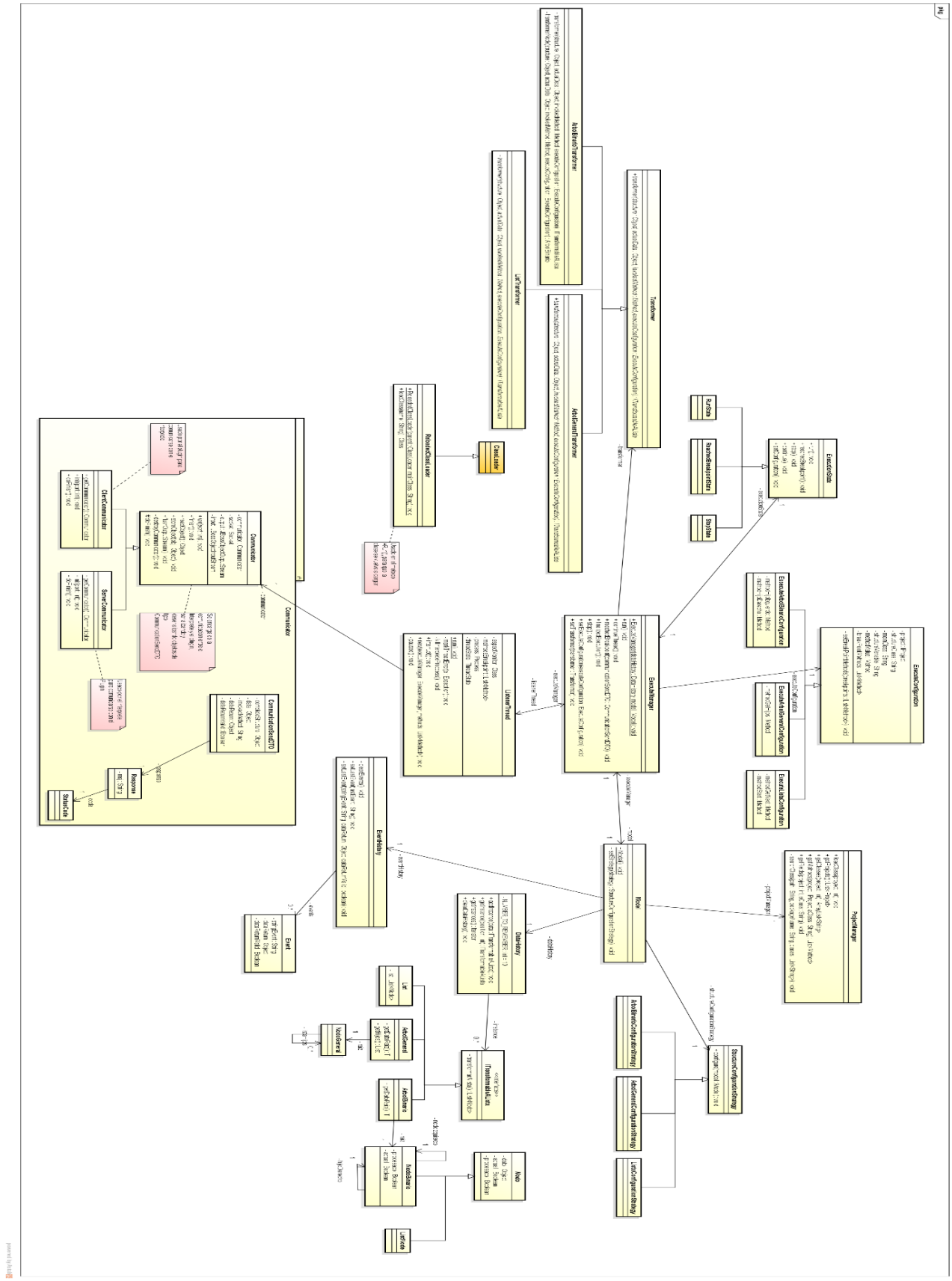

Figura 36 Modelo UML de clases

- ExecuteManager: Esta clase permite el control de la ejecución, es el punto de entrada a cualquier tarea relacionada con la puesta en marcha del debug.

- ExecuteConfiguration: Este objeto POJO contiene la configuración seleccionada por el alumno para ejecutar a continuación.

- ProjectManager: permite realizar las tareas relacionadas a los proyectos del alumno. Ya sea cargar sus clases, recorrerlas (Reflection), etc.

- Transformer: Clase Abstracta que posee un único método, el cual se encarga de descomponer el objeto de los alumnos e introducirlos en una estructura similar propia (Que implementa la interfaz TansformableALista) para su mejor recorrido. La idea es que según la estructura que el alumno desee inspeccionar, se tenga un Transformer diferente.

- TansformableALista: Permite transformar una estructura a su equivalente en una Lista. Lo cual facilita reconocer los nodos que fueron procesados en instancias anteriores, y poder marcarlos en las nuevas instancias.

- ArbolBinario, NodoBinario: Implementación propia de la estructura. La idea es que el ArbolTransformer obtenga la estructura de los alumnos (Árbol) y con reflection construya un ArbolBinario para facilitar su uso posterior acceso. NodoBinario también posee información sobre cómo el árbol ha sido recorrido.

- DataHistory: Mantiene un historial de las instancias.

- EventHistory: Mantiene el historial de eventos.

- ListenerThread: Thread que maneja la interacción con el proceso Intérprete. Este se queda esperando mensajes y, tras la llegada de alguno, informa al thread de la UI.

- Communicator: Administra el uso de los Sockets y de los Stream para la comunicación entre el ListenerThread y el Intérprete, en ambos sentidos (ClientCommunicator y ServerCommunicator)

- CommunicationSendDTO: clase POJO que envía el Intérprete al ListenerThread, con la información del evento ocurrido en la ejecución y datos asociados (Breakpoint, error o finalización exitosa)

- ReloadedClassLoader: implementación propia de un classloader que permite la carga repetida de clases. Es necesario debido a que los alumnos estarán modificando sus

clases entre las distintas ejecuciones, pero el Eclipse no se cierra y por ende no descarga las clases ya cargadas, por lo que no se reflejarán los cambios.

7.3. Diagramas de secuencias

En el siguiente diagrama de secuencia, se muestra como comienza la ejecución de una sesión de debug, así como también, el momento de instanciación del thread que dispara el proceso intérprete y maneja la comunicación con el mismo.

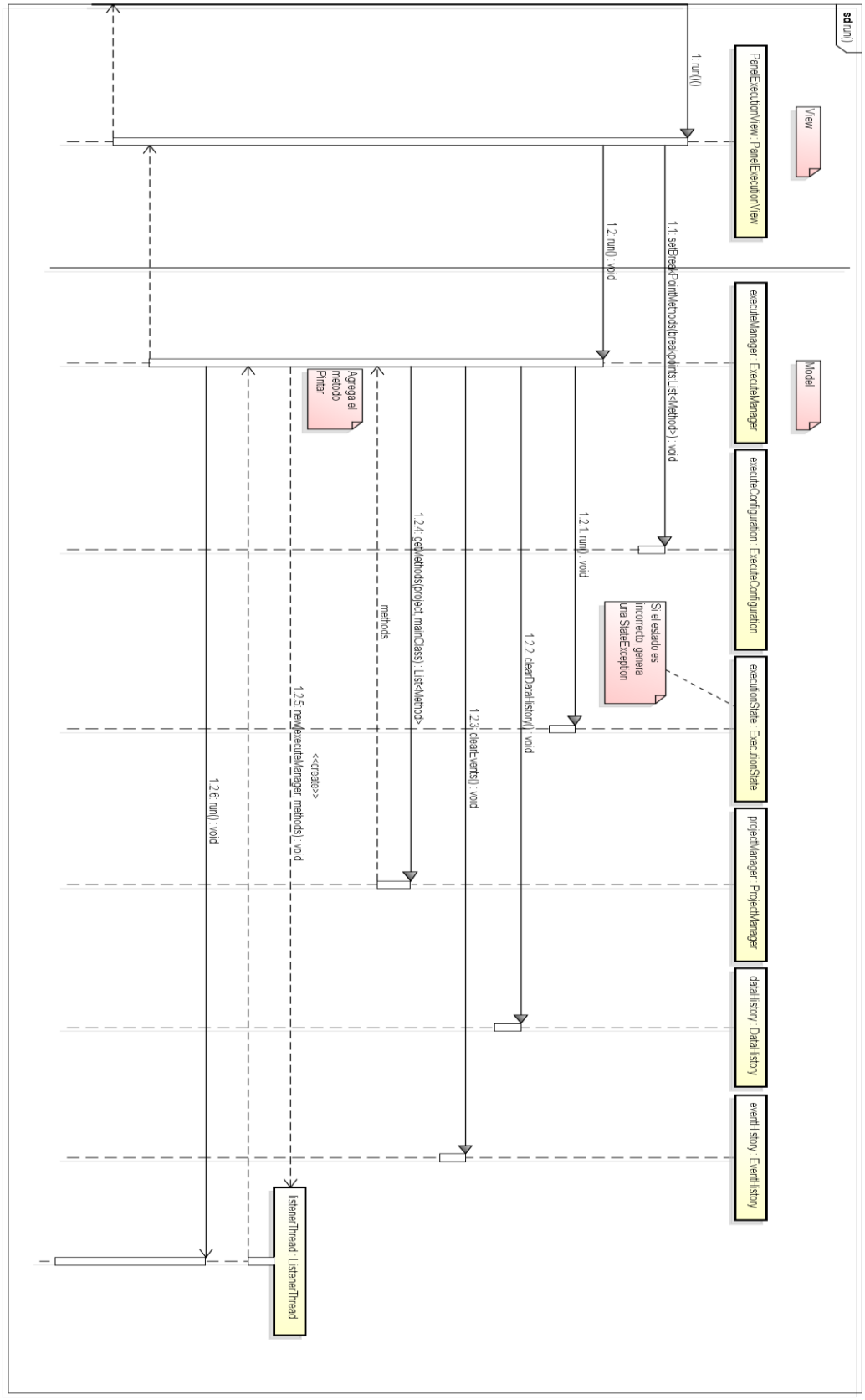

Figura 37 Diagrama UML de secuencia - Run()

Capítulo 8 - SWT

Es un conjunto de componentes para construir interfaces gráficas en Java, recupera la idea original de la biblioteca AWT de utilizar componentes nativos, con lo que adopta un estilo más consistente en todas las plataformas, pero evita caer en las limitaciones de ésta.

Los componentes poseen clases Java e interfaces que permiten el desarrollo de interfaces gráficas [13] [14]. Tales librerías utilizan "*widgets*" nativos de la plataforma siempre que sea posible, accediéndolos a través del Framework JNI[2]. Se encuentran alojados en los paquetes "*org.eclipse.swt.widgets"* y "*org.eclipse.swt.custom".*

8.1. Construyendo Interfaces de Usuario con SWT

Cada Sistema Operativo contiene interfaces gráficas para realizar su propia interfaz. El objetivo de SWT es darles a los programadores Java, acceso directo a estos componentes y de esta manera configurarlos y posicionarlos de la forma que se desee.

Un aspecto importante de este conjunto de librerías, es que no debemos preocuparnos por el Sistema Operativo del usuario final; el funcionamiento de un botón SWT en Windows será el mismo para Windows; en Linux, se comportará como un botón Linux, y así con los diferentes Sistemas Operativos. [15].

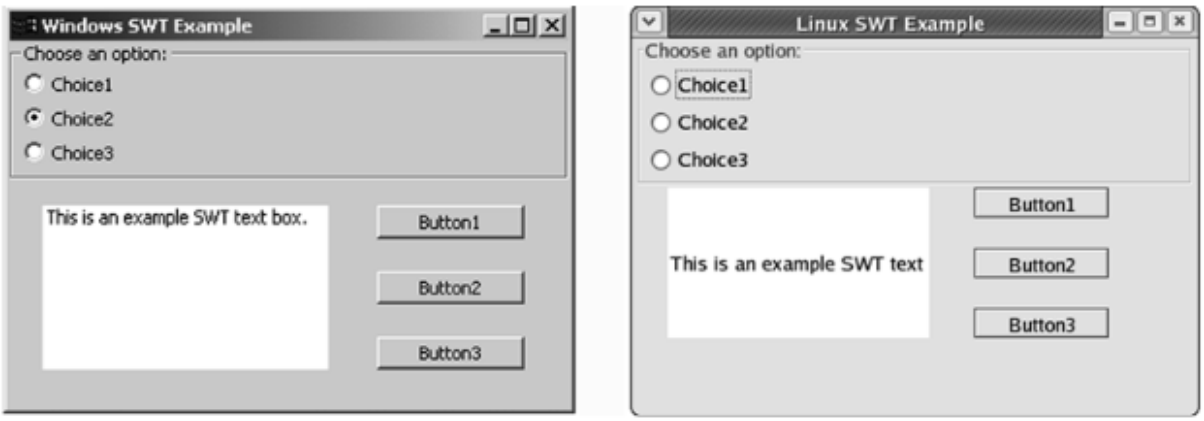

Figura 38 Ejemplo de SWT

En adición a los componentes gráficos, SWT nos provee de acceso a eventos. Por ejemplo, podemos detectar que botones ha presionado el usuario y realizar algún comportamiento al respecto.

[2] Es un framework que permite que el código Java llame y sea llamado por código nativo de aplicaciones escritas en otros lenguajes como C, C++ y Assembler.

Finalmente, si queremos agregar gráficos a nuestra aplicación, nos brinda de una gran cantidad de herramientas para creación de imágenes, trabajar con nuevas fuentes como también dibujar nuevas figuras.

8.2. Programando en SWT

Listing 2.1 HelloSWT.java

```
package com.swtjface.Ch2;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

public class HelloSWT
{
  public static void main (String [] args)
  {
    Display display = new Display();
    Shell shell = new Shell(display);

    Text helloText = new Text(shell, SWT.CENTER);
    helloText.setText("Hello SWT!");
    helloText.pack();

    shell.pack();
    shell.open();
    while (!shell.isDisposed())
    {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

❶ Allocation and initialization

❷ Adding widgets to the shell

❸ GUI operation

Figura 39 Codigo de ejemplo de SWT

1. La primera parte comienza creando una instancia de las clases Display y Shell. Esto permite a la GUI acceder a los recursos de la plataforma subyacente.

2. La siguiente sección agrega un Text widget a la shell. Las aplicaciones normalmente se componen de varios widgets y de grupos de los mismos (en containers) que son agregados como hijos del "*shell*". Los "listener" y "events" se encuentran definidos para cada uno de ellos con el cual el usuario pueda actuar sobre él.

3. La última parte representa la operación de la GUI. Hasta este punto, toda la aplicación no hace nada más que inicializar variables. Pero cuando el método *open()* de la

clase Shell es invocado, la ventana principal de la aplicación toma forma y sus hijos son renderizados en la pantalla. Mientras el Shell permanezca abierto, la instancia de Display usa su método *readAndDispatch()* para realizar un seguimiento de los eventos del usuario que son relevantes en la cola de eventos de la plataforma. Cuando una de estas acciones involucra cerrar la ventana, los recursos asociados al objeto Display son liberados.

Para el desarrollo de la parte visual del plugin, se utilizó SWT debido a diversos problemas encontrados con la tecnología Swing[3]. Los problemas radicaban en anomalías al momento de dibujar el árbol, donde había ciertas funciones que generaban incompatibilidades con la vista, por lo que ocasionaban conflictos.

La vista principal de la herramienta (General View) está compuesta por varios de estos componentes (línea 62 - 75).

```
52 public class GeneralView extends ViewPart implements Observer {
53
54     /**
55      * The ID of the view as specified by the extension.
56      */
57     public static final String ID = "com.fl.algolipse.plugin.views.Algolipse";
58
59     protected Logger log = Logger.getLogger(this.getClass());
60     private Model model;
61     int instanceNumber = 0;
62     private ExpandBar expandBar;
63     private Composite panelUltimaInstancia;
64
65     private Label labelLastEvent;
66     private Button btnLog;
67     public  Display display;
68     public  Shell shell;
69     private Composite composite_1;
70     private Label lblNewLabel_1;
71     private Label lblNewLabel_2;
72     private Label lblProcesado;
73     private Label lblSinProcesar;
74     private Label lblNuevoNodo;
75     private ScrolledComposite scrolledComposite;
```

Figura 40 Componentes de la clase GeneralView

Como se muestra en la línea 62, se define un componente ExpandBar[4], el cual se encargará de contener las instancias previas de la estructura.

El Composite[5] panelUltimaInstancia definido en la línea 63, es un contenedor que se ocupa de llevar la última instancia de la estructura recursiva.

[3] Biblioteca gráfica para Java. Incluye widgets para interfaz gráfica de usuario tales como cajas de texto, botones, desplegables y tablas

[4] Componente SWT que posee uno o más ExpandItems formando una especie de acordeón.

```
252                 /*
253                  * En el momento en que se ejecuta el thread, los compenentes de la vista pueden estar "Disposed"
254                  * por su ejecución asincronica.
255                  */
256                 log.info("Va a dibujar Arbol");
257                 if(expandBar!= null && !expandBar.isDisposed()){
258                     ArbolExpresionGrafico aGrafico = new ArbolExpresionGrafico(
259                             expandBar, SWT.NONE, model.getDataHistory()
260                                     .getLastInstance());
261                     GridLayout layout = new GridLayout ();
262                     layout.marginLeft = layout.marginTop = layout.marginRight = layout.marginBottom = 10;
263                     layout.verticalSpacing = 10;
264                     aGrafico.setLayout(layout);
265                     instanceNumber++;
266                     ExpandItem item0 = new ExpandItem(expandBar, SWT.NONE, 0);
267                     item0.setText( model.getEventHistory().getLastEvent().toString());
268                     item0.setHeight( 300 );
269                     item0.setControl( aGrafico );
270                     log.info("Item de expandbar dibujado");
271                 }
272 
273                 for (Control elements : panelUltimaInstancia.getChildren()){
274                     elements.dispose();
275                 }
276 
277                 ArbolExpresionGrafico aGrafico = new ArbolExpresionGrafico(
278                         panelUltimaInstancia, SWT.NONE, model.getDataHistory()
279                                 .getLastInstance());
280                 GridLayout layout = new GridLayout ();
281                 layout.marginLeft = layout.marginTop = layout.marginRight = layout.marginBottom = 10;
282                 layout.verticalSpacing = 10;
283                 aGrafico.setLayout(layout);
284                 panelUltimaInstancia.layout();;
285                 aGrafico.layout();;
286 
```

Figura 41 Nueva instancia de la estructura en la vista

A su vez se creó un contenedor propio (línea 258) donde se va a generar la estructura (ArbolExpresionGrafico).

Los componentes ExpandItem son generados en la línea 266 y tendrán la función de contener el estado del ArbolExpresionGrafico dependiendo la instancia de la recursión que se encuentre. Es decir, se va a generar uno nuevo por cada instancia de la recursión.

A partir de la línea 277, se agrega al panelUltimaInstancia, encargado de llevar la última instancia de la recursión del ArbolExpresionGrafico.

[5] Contenedor SWT de componentes. Es el “acrónimo” JPanel de Swing.

Capítulo 9 - Programación orientada a Aspectos

9.1. Vida sin POA

Los diferentes paradigmas de programación: funcional, procedural y sobre todo el orientado a objetos, ofrecen potentes mecanismos para separar intereses, sobre todo los relacionados con la lógica del negocio de la aplicación, pero no ofrecen tan buenos resultados a la hora de tratar otros intereses que no pueden ser encapsulados en una única entidad puesto que “atraviesan” diferentes partes del sistema. A este tipo de intereses se les denomina intereses transversales (crosscutting concern).

En un sistema puramente orientado a objetos, usualmente se agrega el código necesario para cada uno de los módulos como se muestra en la siguiente figura. Es decir, diferentes módulos en un sistema implementan inquietudes transversales y centrales habiendo código disperso.

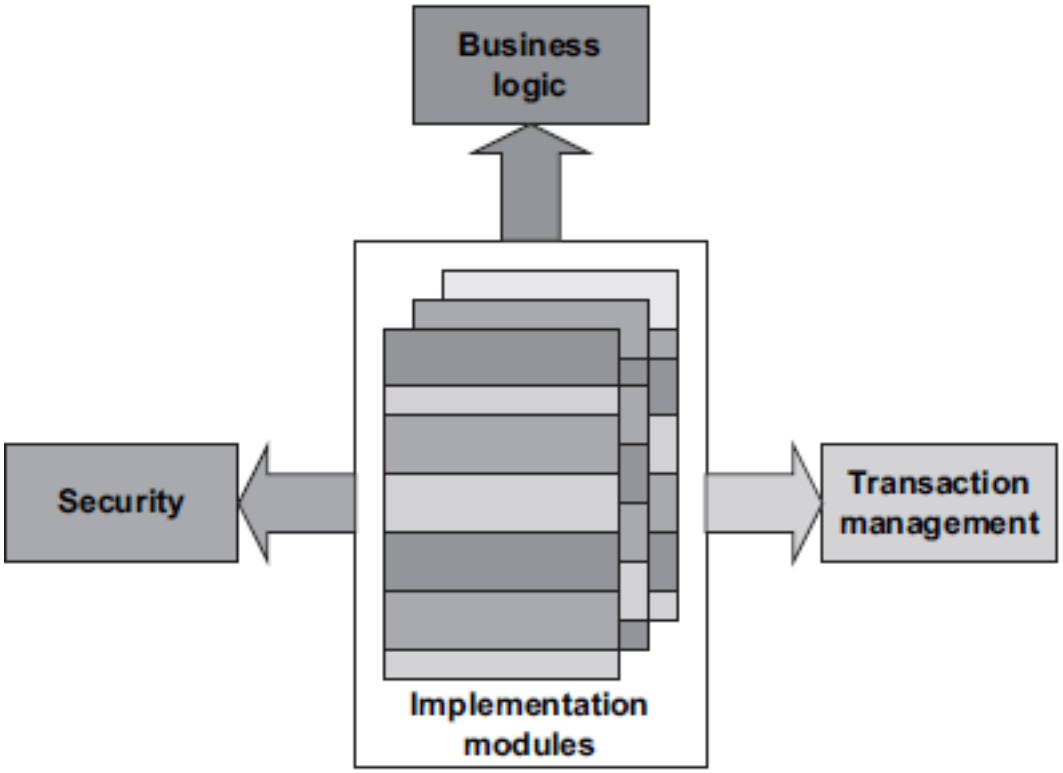

Figura 42 Ejemplo de una estructura sin POA

El principal problema que persigue se establece cuando se debe realizar alguna modificación, ya que la misma tiene que realizarse en todos los módulos que se encuentran afectados por tal suceso. Es decir, tendremos un alto costo a la hora de implementar las nuevas características.

9.2. Posibles problemas

Código entretejido (Weaving code)

Es causado cuando un módulo es el encargado de implementar múltiples “concerns (intereses)” simultáneamente. Tales intereses pueden ser: lógica de negocio, performance, sincronización, seguridad, etc.

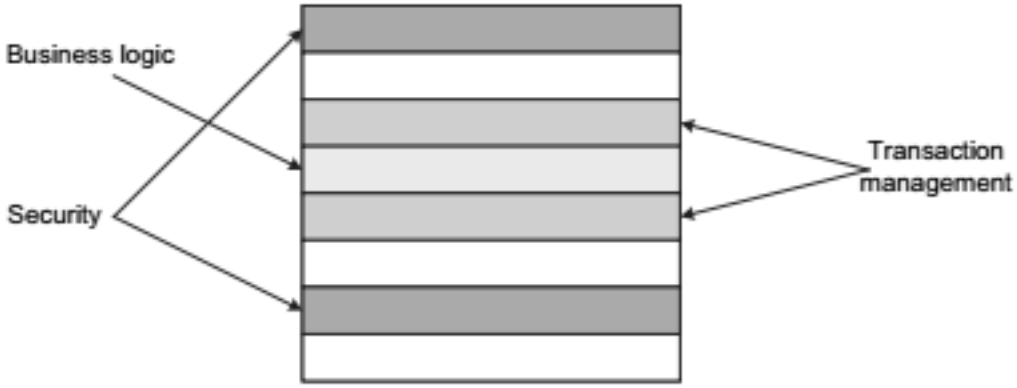

Figura 43 Modelo con código enredado

Código Disperso

Es causado cuando una simple funcionalidad es implementada en múltiples módulos. Por ejemplo, en un sistema que utiliza una base de datos, los problemas de rendimiento pueden afectar a todos los módulos de acceso a base de datos.

La siguiente Figura 43 nos muestra cómo un sistema bancario implementa la seguridad utilizando tecnologías convencionales. Incluso cuando se utiliza un módulo de seguridad bien diseñado, que ofrece una API abstracta y oculta los detalles, cada cliente “the accounting module, the ATM module, and the database module“ aún necesitan el código para invocar la API de seguridad para comprobar el permiso.

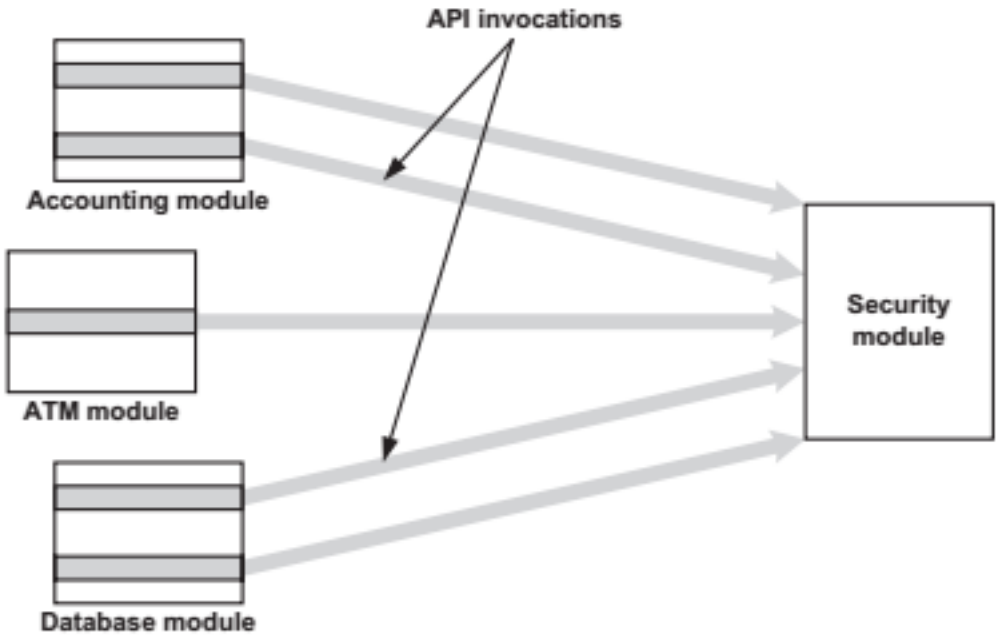

Figura 44 Código disperso

9.3. Vida con POA

Nos permite capturar los intereses que atraviesan el sistema en entidades bien definidas llamadas aspectos, consiguiendo así una clara separación de los mismos.

Usando programación orientada a aspectos, ninguno de los módulos tiene una llamada a la API de seguridad. A diferencia del esquema anterior, estarán integrados en el "interés" de seguridad y el aspecto de seguridad.

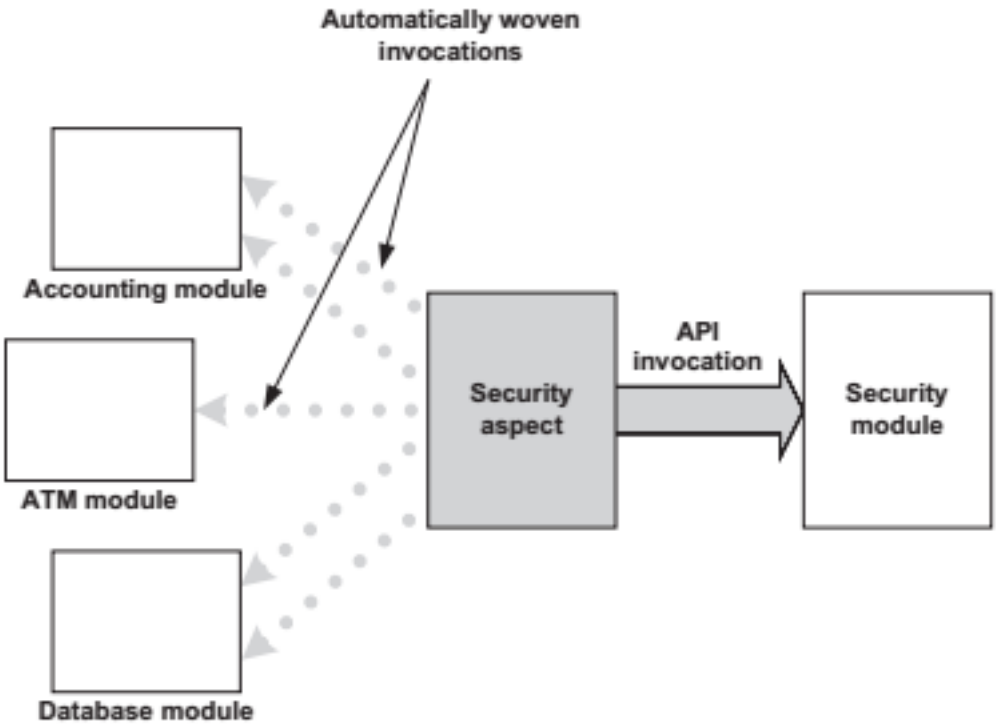

Figura 45 Ejemplo de una estructura con POA

9.4. Diseño orientado aspectos

Etapa 1: Identificar competencias/intereses (concern)

Consiste en descomponer los requisitos del sistema en intereses y clasificarlas en:

1. Intereses centrales (core-concern): los que están relacionados con la funcionalidad central del sistema, de carácter funcional. Son las que Gregor Kiczales[6] denomina componentes [16].

2. Intereses transversales (crosscutting-concern): los que afectan a varias partes del sistema, relacionadas con requerimientos no funcionales del mismo, normalmente de carácter no funcional. Son las que Gregor Kiczales denomina aspectos [6].

Etapa 2: Implementar competencias/intereses

Consiste en implementar cada interés independientemente: Para implementar los intereses básicos usaremos el paradigma que mejor se ajuste a ellos (POO, programación procedural o funcional), y dentro del paradigma, el lenguaje que mejor satisfaga las necesidades del sistema. A este lenguaje lo denominaremos lenguaje base. Para implementar los intereses transversales usaremos uno o varios lenguajes orientados a

[6] Uno de los encargados de desarrollar el Lenguaje AspectJ.

aspectos, de propósito específico o general, encapsulando cada competencia en unidades llamadas aspectos. Estos lenguajes orientados a aspectos deben ser compatibles con el lenguaje base para que los aspectos puedan ser combinados con el código que implementa la funcionalidad básica y así obtener el sistema final. Normalmente estos lenguajes suelen ser extensiones del lenguaje base, como es el caso de AspectJ (Extensión del Lenguaje Java).

Etapa 3: Componer el sistema final

La implementación del sistema final se conoce como entretejido (weaving) o integración. Consiste en combinar los aspectos con los módulos que implementan la funcionalidad principal del sistema dando lugar al sistema final. El módulo encargado de realizar este proceso recibe el nombre de tejedor de aspectos (aspect weaver) y hace uso de unas reglas de entretejido para llevar a cabo el proceso.

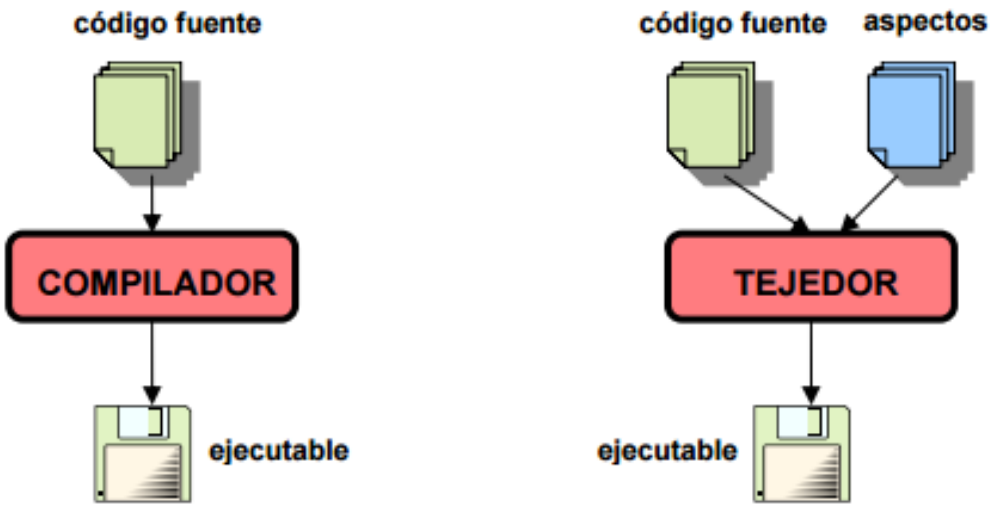

Figura 46 Lenguaje sin POA vs Lenguaje con POA

9.5. Lenguaje Orientado a Aspectos

Los lenguajes orientados a aspectos suelen basar sus reglas de entretejido en el concepto de punto de enlace (join point). Un punto de enlace es un lugar bien definido en la ejecución de un programa. Mediante las reglas de entretejido indicaremos aquellos puntos de enlace que se ven afectado por cierto aspecto.

Las construcciones del lenguaje para definir las reglas de entretejido deben ser potentes y expresivas, permitiendo expresar desde reglas muy específicas que involucren a uno o pocos puntos de enlace, hasta reglas muy genéricas que engloben a un gran número de puntos. Estos mecanismos y construcciones varían de unos lenguajes a otros.

9.6. El Modelo Punto de enlace (Join Point)

El punto de enlace es el concepto central de la Programación Orientada a Aspectos. Consiste de dos partes: el punto de enlace propiamente dicho, el cual es un punto candidato en la ejecución de una aplicación donde el aspecto puede ser conectado (Esto puede ser un método que está siendo llamado, una excepción que está siendo disparada o cualquier campo que se esté modificando).

El lenguaje AspectJ para la definición de puntos de enlace, es sofisticado, expresivo y elegante; nos brinda la posibilidad de seleccionar los puntos de enlace basados en información estructural tales como tipos, nombres y argumentos, así como condiciones en tiempo de ejecución para realizar un control del flujo. Es decir, nos permite seleccionar exactamente el punto donde necesitamos implementar una funcionalidad que se encuentra dispersa en diferentes partes del código.

Consideremos una situación donde necesitemos implementar gestión de transacciones y debido a la naturaleza transversal, aspectos es el enfoque preferido. Lo primero que se tiene que realizar es identificar los lugares donde se necesita comenzar una transacción como también los lugares donde se realiza el cometido o la vuelta atrás de la misma antes de que la lógica finalice.

Una vez identificado los sectores de importancia, se debe escribir un punto de corte que seleccione los puntos de enlace requeridos. Para la gestión de transacciones es probable que el punto de enlace sea la ejecución de los métodos; lo que se debe realizar es tratar de identificar alguna coincidencia entre ellos. Para el ejemplo que se está presentando, puede ser que varios métodos tengan una anotación @Transactional para realizar la transacción y de esta manera tomar esa coincidencia para realizar el punto de corte.

Por último, se construye un Aspecto, que encapsule la funcionalidad de la gestión de transacciones. Es el Aspecto el que se encarga de usar el punto de corte escrito anteriormente para comenzar a realizar el cometido o la vuelta atrás de la transacción en el punto de enlace seleccionado.

9.7. Implementación de puntos de corte básicos

El lenguaje para la realización de puntos de corte, utiliza la misma sintaxis que “AspectJ”. Pueden ser declarados dentro de un aspecto, una clase o una interface, y al igual que con los datos y métodos, se puede usar un especificador de acceso para restringir el acceso a los mismos.

9.8. Implementación del Lenguaje Orientado a Aspectos

La implementación de un lenguaje orientado aspectos debe proporcionar una entidad, el tejedor de aspectos, que se encargue de integrar los aspectos con el código base en un proceso que se denomina entretejido (weaving). Existen dos tipos diferentes de entretejido:

Entretejido estático

El tejedor genera un nuevo código fuente como resultado de integrar el código de los aspectos en los puntos de enlace correspondientes del código base. En algunos casos el tejedor tendrá que convertir el código de aspectos al lenguaje base, antes de integrarlo. El nuevo código fuente se pasa al compilador del lenguaje base para generar el ejecutable final. Es el entretejido usado por AspectJ 1.0.x. Otra posibilidad es que el entretejido tenga lugar a nivel de byte-code[7], es decir, el compilador de AspectJ entreteje los aspectos en los ficheros .class de nuestra aplicación, generando los .class de salida correspondientes. Es el entretejido usado por AspectJ 1.1.x.

Las ventajas del entretejido estático son:

- El entretejido se realiza en tiempo de compilación, por lo que se evita un impacto negativo en la eficiencia de la aplicación al no existir sobrecarga en la ejecución. Mayor nivel de fiabilidad al realizarse todas las comprobaciones en tiempo de compilación, evitando así posibles errores durante la ejecución.

- Son más fáciles de implementar y consumen menos recursos.

Las desventajas del entretejido estático son:

- Al ser muy difícil identificar los aspectos en el código entretejido, es casi imposible poder modificarlos en tiempo de ejecución, permaneciendo fijos durante todo el ciclo de vida del programa. Tampoco es posible añadir o eliminar aspectos dinámicamente.

[7] Código intermedio más abstracto que el código máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable

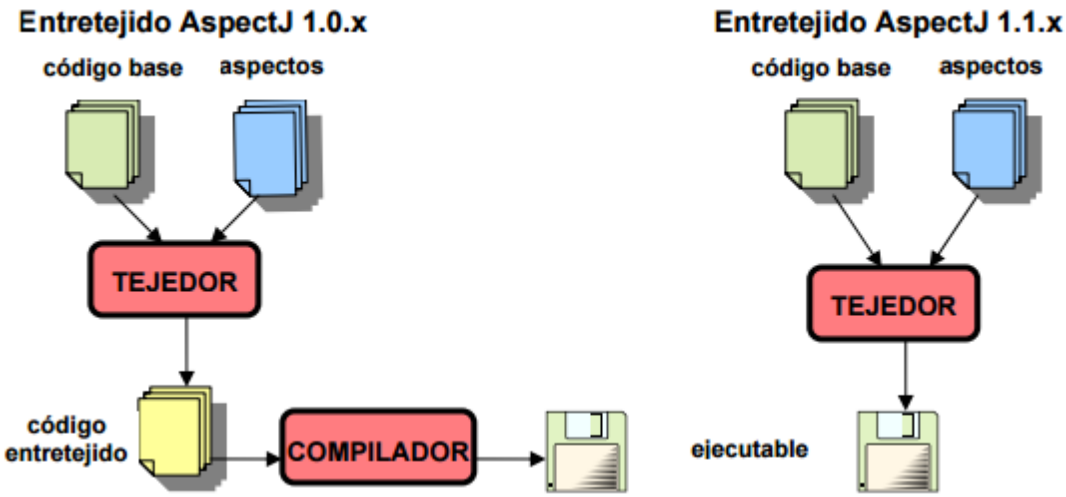

Figura 47 (Weaving estático) Diferencia entra AspectJ 10 y 1.1

Entretejido dinámico

Para poder realizar entretejido dinámico es necesario que tanto los aspectos como sus reglas de entretejido, sean modeladas como objetos y estén disponibles en tiempo de ejecución. La implementación del lenguaje orientado a aspectos proporcionará un entorno de ejecución controlado donde cada vez que se llega a un punto de enlace ejecutará el código del aspecto o aspectos que lo afectan. Mediante este tipo de tejedores si es posible añadir, modificar y eliminar aspectos en tiempo de ejecución.

La ventaja principal del entretejido dinámico frente al estático es que permite añadir, modificar o eliminar aspectos dinámicamente, en tiempo de ejecución, aumentando las posibilidades del programador. Por otra parte, nos encontramos con ciertas desventajas:

- Existe sobrecarga en la ejecución del programa, al realizar el entretejido en tiempo de ejecución, por lo que disminuye su rendimiento.
- Disminuye la fiabilidad de la aplicación, ya que se pueden producir errores como borrar el comportamiento de un aspecto que posteriormente será invocado.
- Son más difíciles de implementar que los tejedores estáticos y consumen muchos más recursos.

A la hora de desarrollar la herramienta, el enfoque estuvo en ver la manera de no intervenir en el código desarrollado por los estudiantes. Es decir, que ellos solamente tengan que definir sus algoritmos y abstraerse del funcionamiento de la herramienta.

La pregunta que nos surgió fue cómo hacer para capturar el método seleccionado por el alumno de manera de poder inspeccionarlo y así realizar el "debugging" del mismo.

La primera aproximación fue tratar de ver cómo funcionaba el debug de eclipse y así poder incorporar el mismo. Esto no fue posible debido a que resultaba muy complejo y trabajaba a muy bajo nivel.

Por tal finalidad se pensó en POA (Programación orientada a Aspecto). De esta manera cuando se alcanza una llamada a un método previamente seleccionado, el aspecto lo detectará e informará la presencia de un "*breakpoint*". Como mencionamos al principio, otro de los puntos que llevaron a utilizar el paradigma, se debe a la libre codificación por parte de los estudiantes, donde no habrá convenciones o anotaciones que hagan más difícil la escritura del algoritmo desarrollado. Es decir, no es necesario que realicen algún tipo de "marca" para indicar de qué tipo de estructura se trata y cuáles son sus métodos de acceso a la información.

Debido a que las clases de los alumnos ya se encuentran compiladas, se tuvo que utilizar entretejido dinámico. Este último ocurre cuando el "*classLoader"* del proceso carga las clases de los alumnos para su ejecución. A su vez, como el punto de enlace se encuentra definido previamente y no se conocen los nombres de las clases y métodos del alumno, se debe englobar todas las posibilidades y, por ende, observarlas (En nuestro desarrollo, el punto de enlace, son todos los métodos de todos los paquetes, sin incluir los del aspecto). Dentro del aspecto, se valida si es un método de interés para el alumno o no. En caso de que lo sea se detectará y realizará el proceso correspondiente.

```
32    pointcut methodsOfInterest(Object o):
33        execution( * *(..)) &&
34        !within(com.fl.algolipse.aspectclass.*) &&
35        target(o);
36
```

Figura 48 AspectJ en Algolipse

En la imagen anterior, podemos observar la firma del punto de corte, la cual indica que AspectJ bloqueara todos los métodos que no formen parte del "package" *com.fl.algolipse.aspectclass.** (Esto evita un loop al capturar todo). Se puede observar, que este recibe un parámetro de tipo Object, el cual representa el objeto encargado de recibir el mensaje que está siendo observado.

En la siguiente imagen, podemos ver que fue utilizado el modo *around* [8]para manejar el método bloqueado por AspectJ.

```
45⊖    Object around(Object o) : methodsOfInterest( o) {
```

Figura 49 Around en AspectJ

[8] Se utiliza para agregar comportamiento antes y después de la invocación del método alcanzado por AspectJ.

Capítulo 10 - Prototipos

A continuación, se mostrará la interfaz del plugin y se explicará para que sirve cada componente.

Si nos dirigimos al menú del Eclipse y presionamos sobre Window → Show View → Other, y buscamos Algolipse, el entorno nos mostrar la vista para abrirla como se muestra en la figura siguiente.

10.1.La vista Algolipse

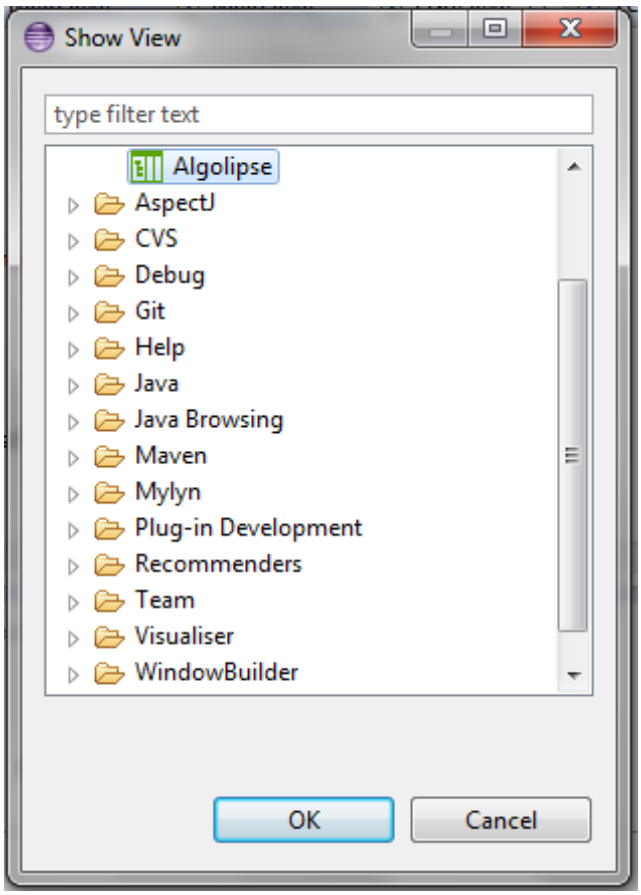

Figura 50: Ventana "Show View" con la vista "Algolipse"

Una vez que se elige la vista, el entorno aparecerá con la vista agregada. Como explicamos con anterioridad, la misma se sitúa dentro de la perspectiva del Eclipse.

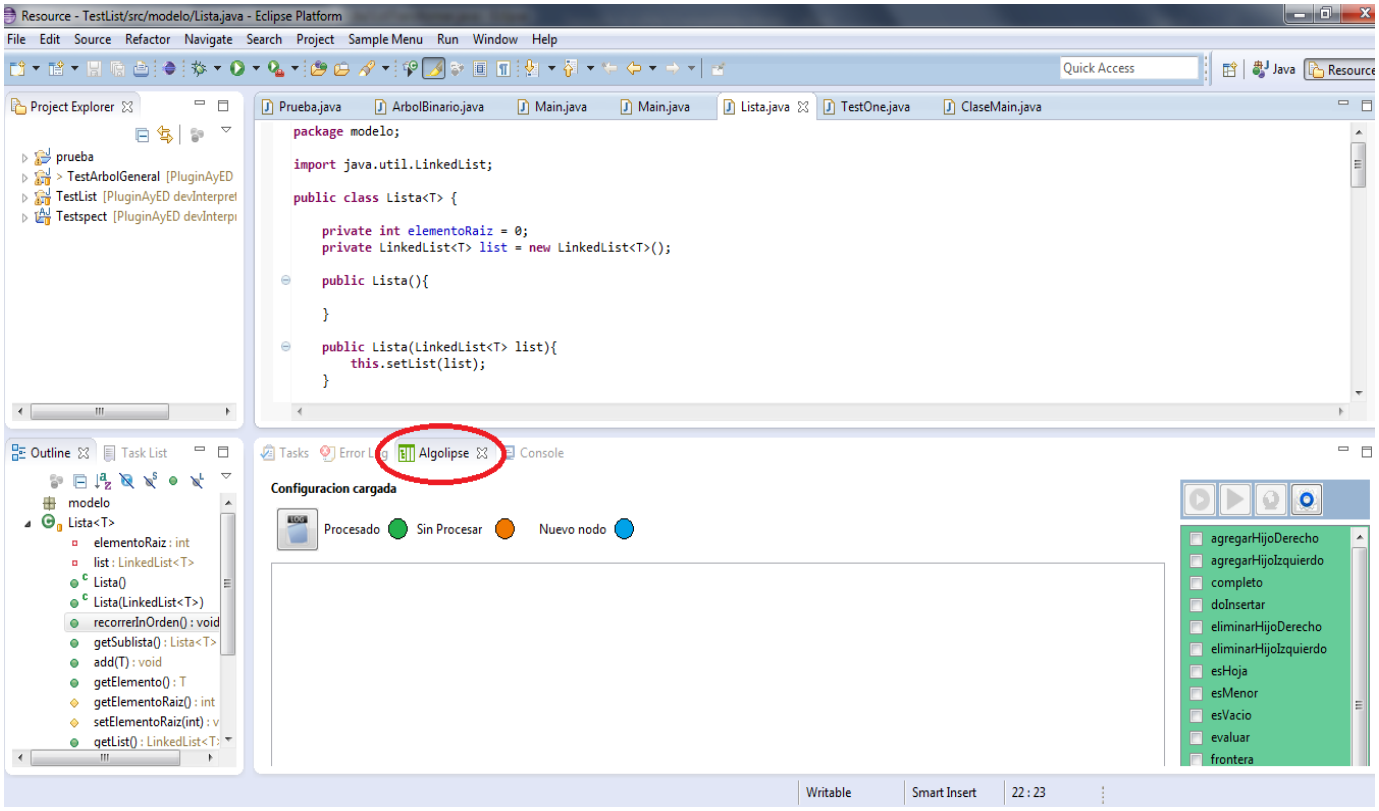

Figura 51: Vista "Algolipse" agregada a la perspectiva Java

La vista Algolipse está compuesta por las siguientes componentes:

1) Panel donde se encuentran los posibles métodos a seleccionar para posteriormente ser inspeccionados.

2) Botonera encargada de comenzar, continuar y parar la ejecución. Además, posee un botón el cual se encarga de abrir la ventana de configuración.

3) Información que le servirá al alumno para ver qué nodos fueron procesados, cuáles son nuevos y cuáles se encuentran sin procesar.

4) Área donde se dibuja la estructura.

5) Pila de invocación a los llamados recursivos.

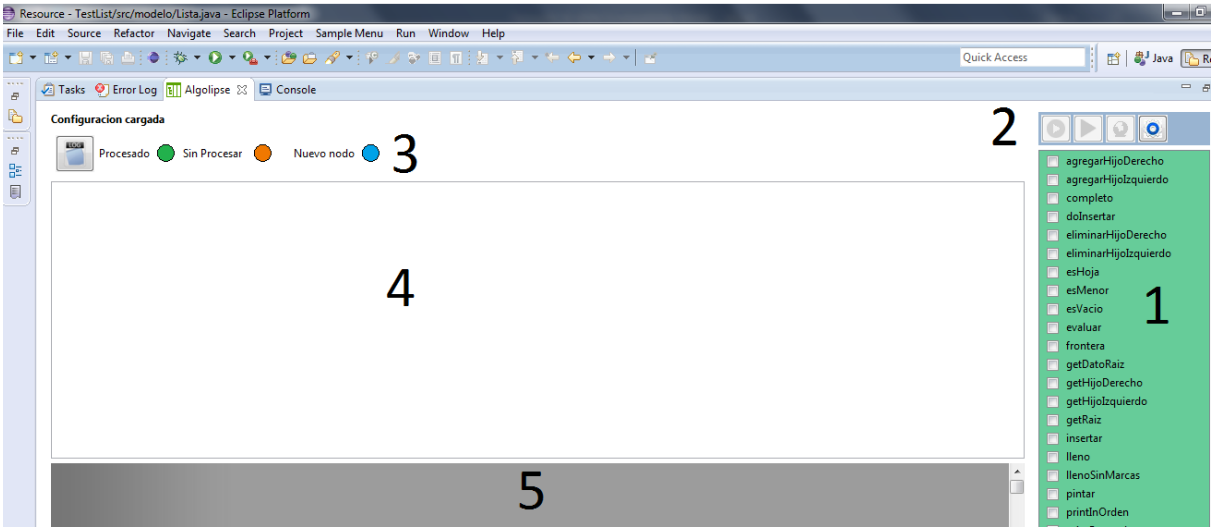

Figura 52: Entorno de "Algolipse"

10.2. Ventana de Configuración

La parte superior, es compartida por todas las estructuras ya que independientemente de la que se elija, se debe seleccionar el proyecto, la clase principal donde el alumno prueba sus ejercicios y la variable de la estructura.

En la parte inferior, se sitúan las estructuras y dependiendo cual se elija, es la información que el alumno debe establecer. Por ejemplo, si se quiere configurar un Árbol Binario, se debe seleccionar la clase encargada de crear un ArbolBinario, el método encargado de devolver el valor de la variable, como también el encargado de retornar el hijo izquierdo y derecho.

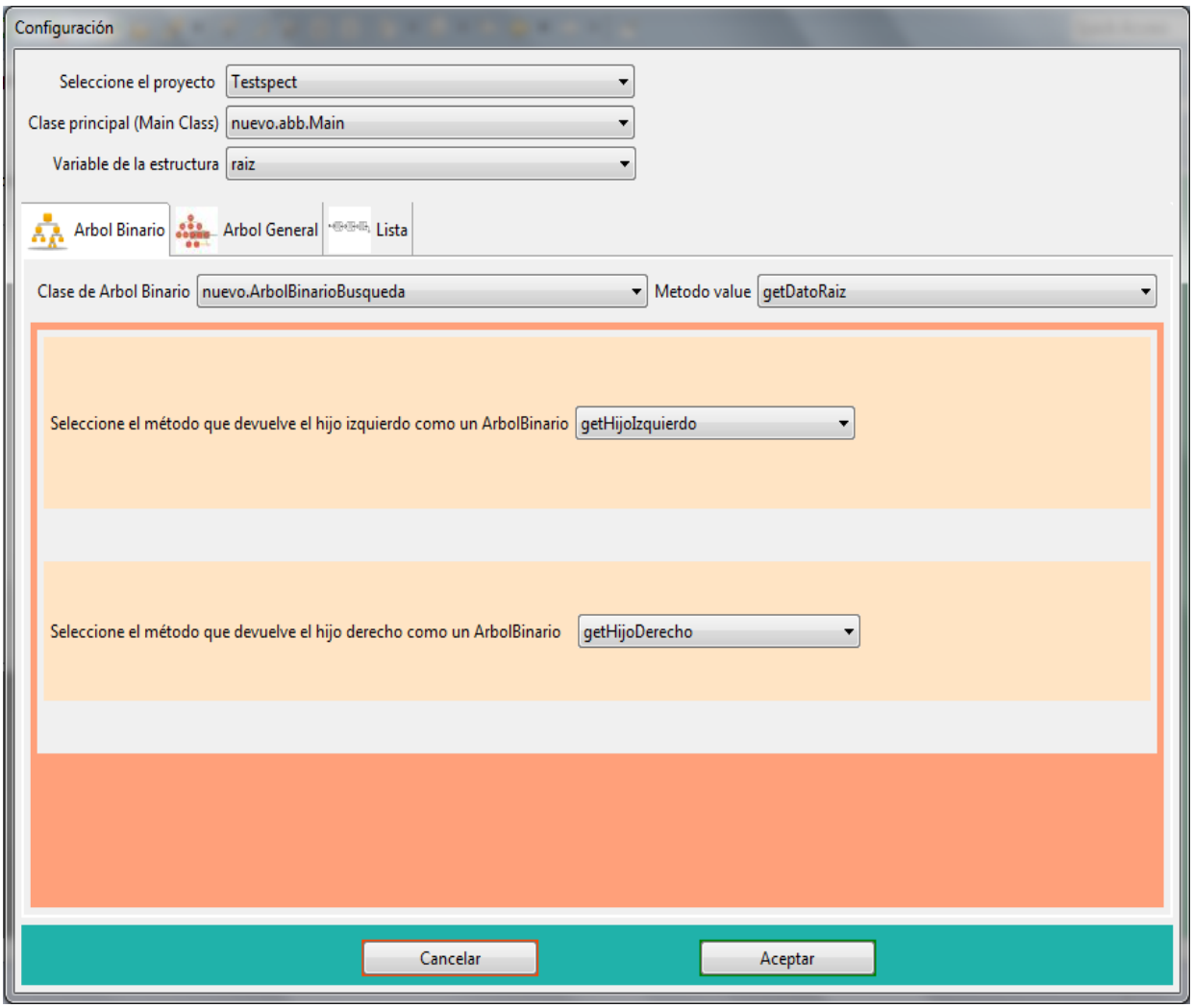

Figura 53: Ventana de configuración

10.3. Ejemplo 1: Ejecución del método “printInOrden()”.

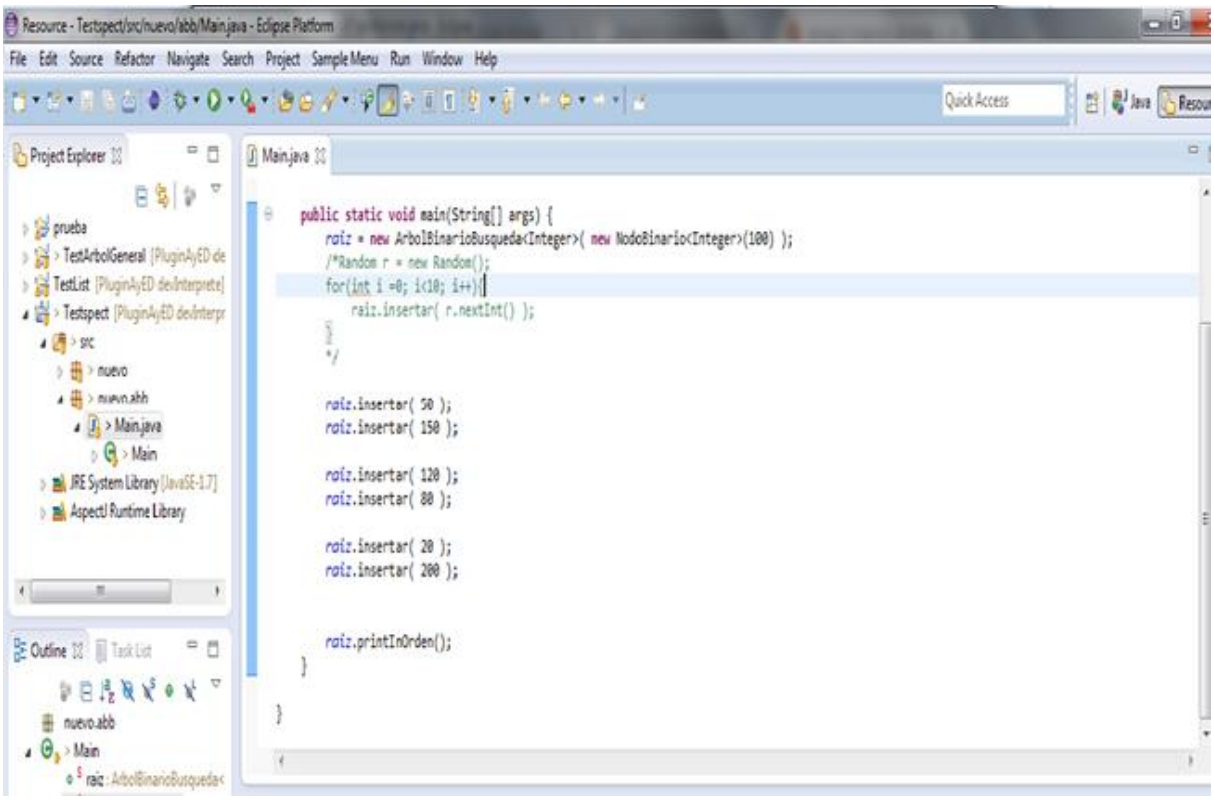

Figura 54: Programa principal para el ejemplo 1

Como se ve en la figura 56, se configura un ArbolBinarioDeBusqueda con los datos 50, 150, 120, 80, 20, 200 y luego se llama al método printInOrden().

Se elige el método printInOrden() de la lista de métodos y se presiona sobre el botón “Play”

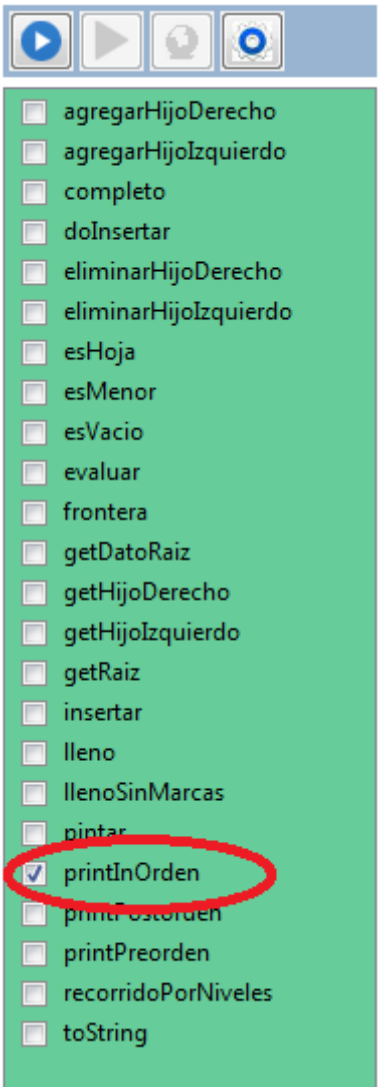

Figura 55: Selección del método a inspeccionar (printInOrden)

A continuación, se muestra el recorrido del árbol binario de búsqueda. Como se ve el nodo 100 se encuentra con un borde negro indicando donde se encuentra situado el algoritmo en ese instante.

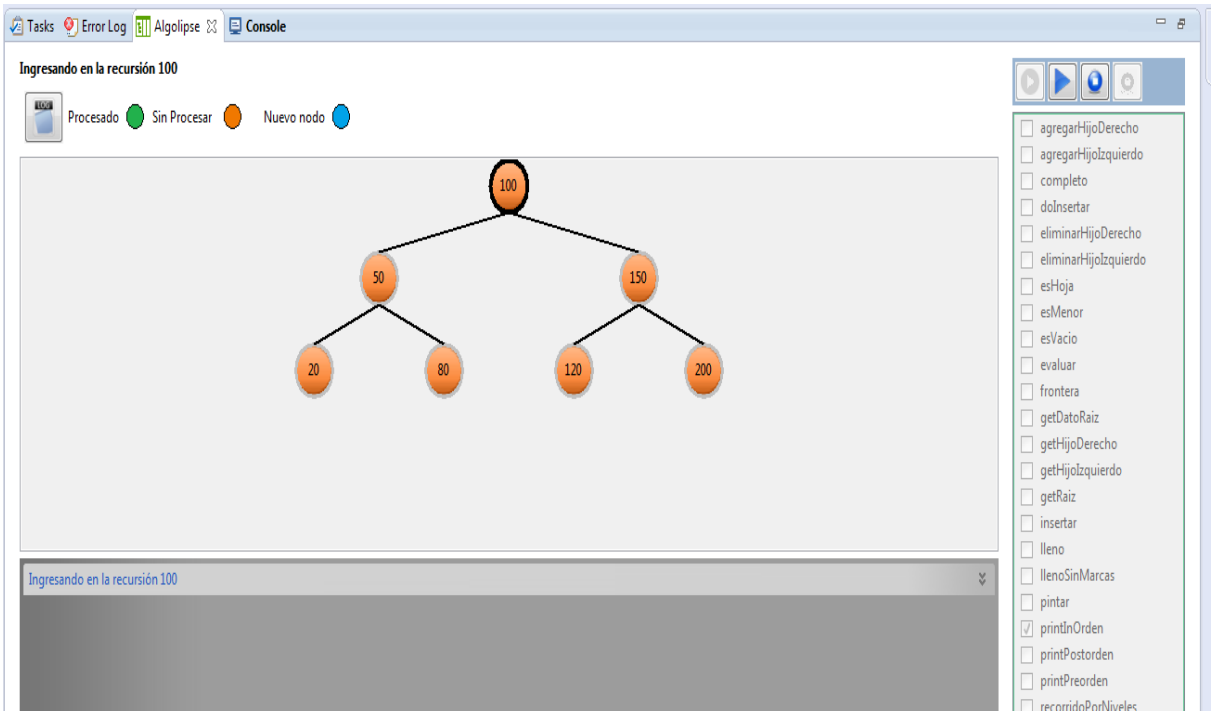

Figura 56: Instancia inicial del recorrido (Nodo 100)

Al presionar sobre el botón siguiente, vemos que el algoritmo avanza quedando situado en el nodo 50.

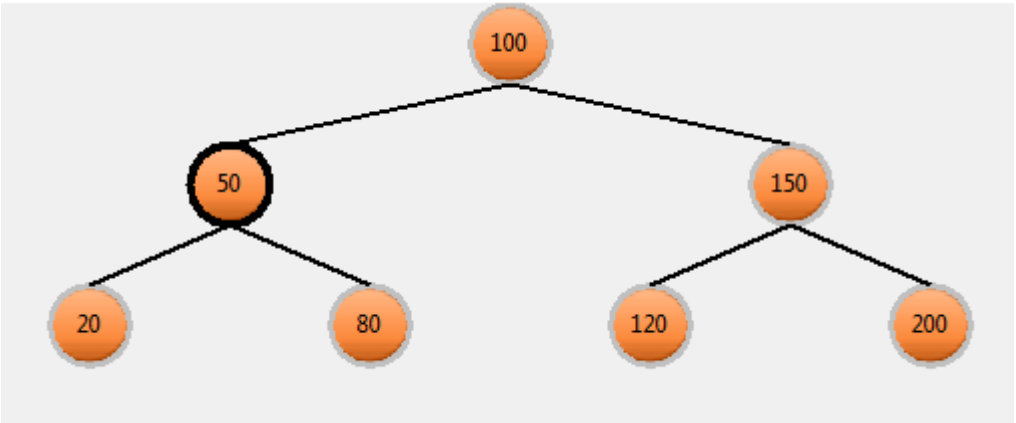

Figura 57: Ingresando en la recursión (Nodo 50)

Si continuamos con la ejecución, el algoritmo continuara avanzando posicionándose en el nodo 20.

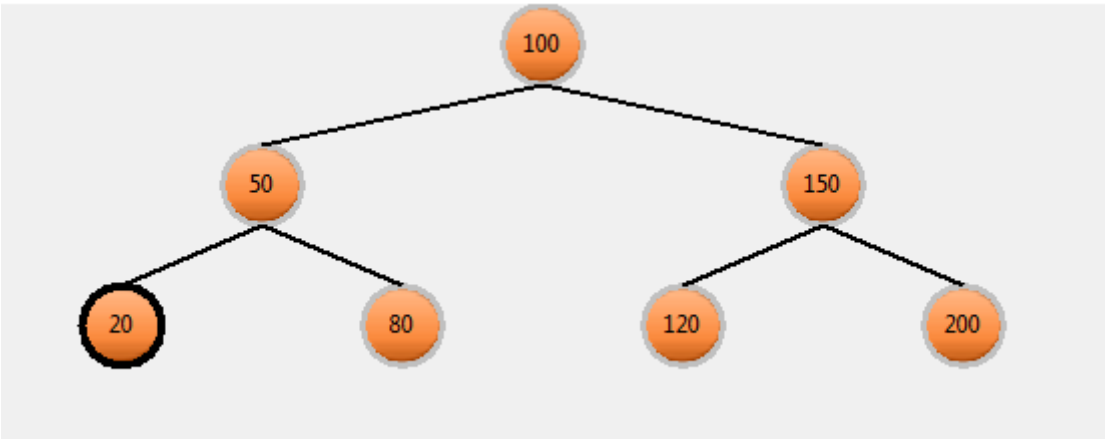

Figura 58: Ingresando en la recursión (Nodo 20)

En la figura 61 se muestra que el nodo 20 ha sido procesado.

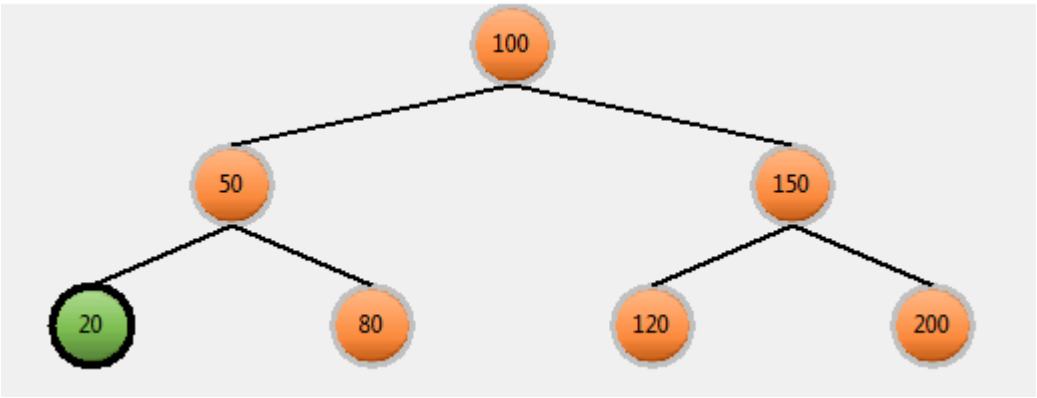

Figura 59: Procesamiento del nodo 20, pintado de color verde

Si seguimos ejecutando vemos que se está volviendo de la recursión.

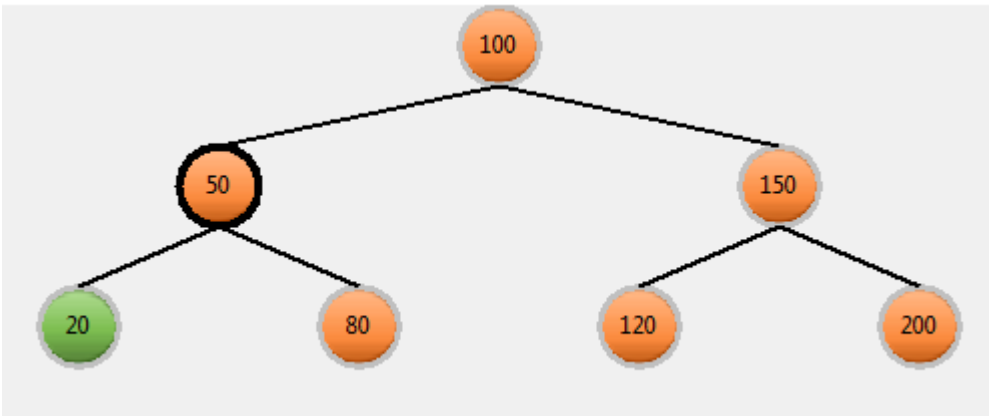

Figura 60: Volviendo de la recursión (Nodo 50)

Si seguimos con la ejecución hasta finalizar el recorrido, nos encontraremos con todo el árbol procesado.

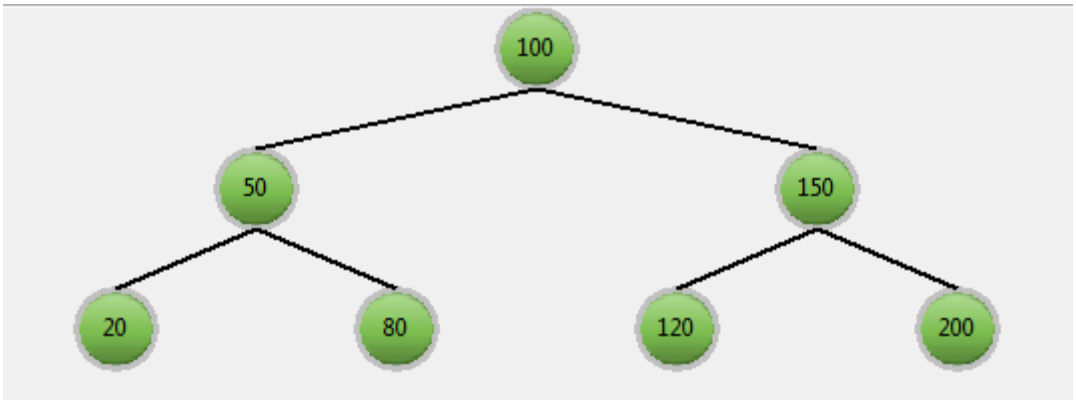

Figura 61: Ejecución finalizada del método "printInOrden()" (Todos los nodos procesados)

10.4. Ejemplo 2: Ejecución del método "evaluar()"

En el siguiente ejemplo, veremos el método evaluar, el cual recorre el árbol de expresión y devuelve, por cada nodo, el resultado de evaluarlo.

En la figura 64, establecemos la configuración de la estructura

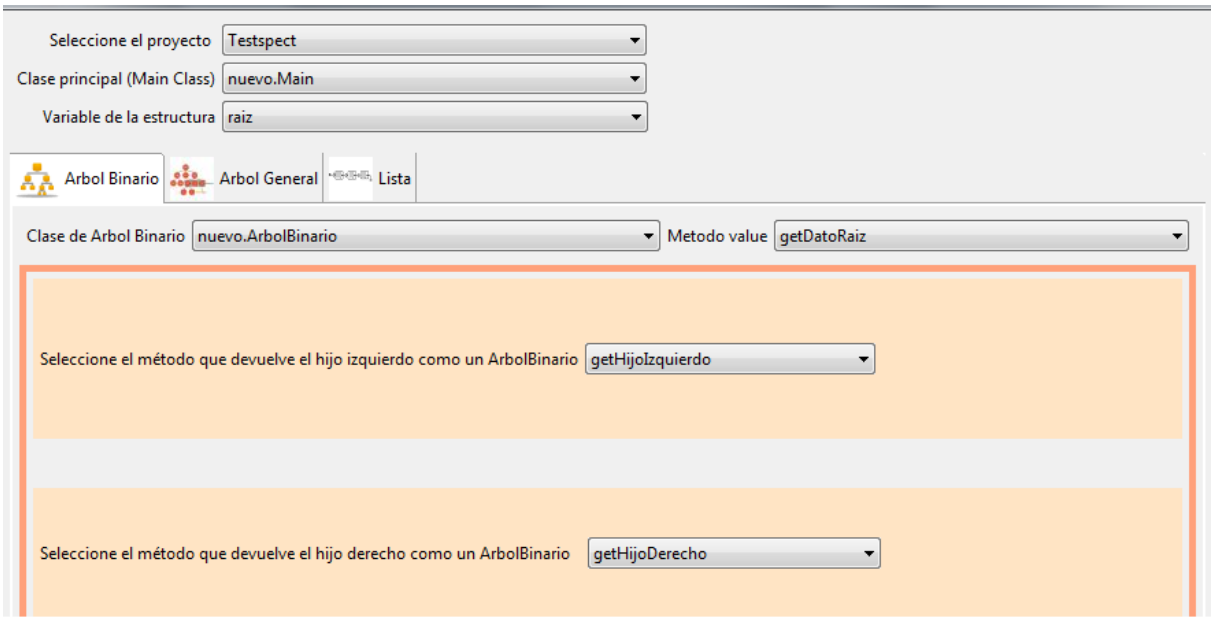

Figura 62: ventana de configuración. Método "evaluar"

En la figura 65, se muestra la selección del método "evaluar" que será analizado

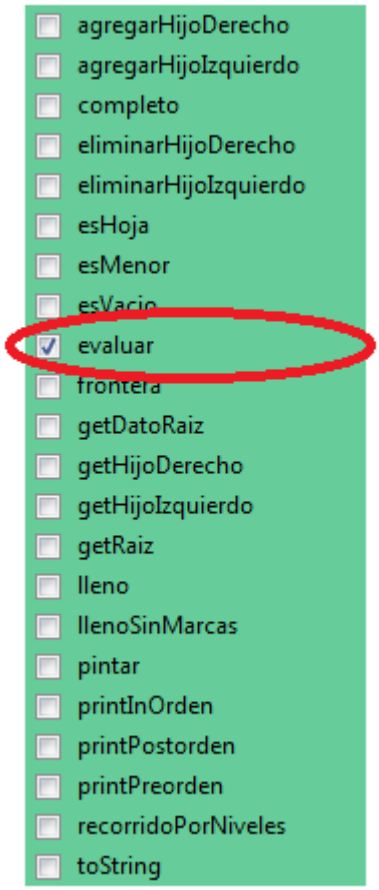

Figura 63: Selección del método a inspeccionar (evaluar)

A continuación, se muestra la primera instancia del árbol. En el cual estamos parados en el nodo raíz.

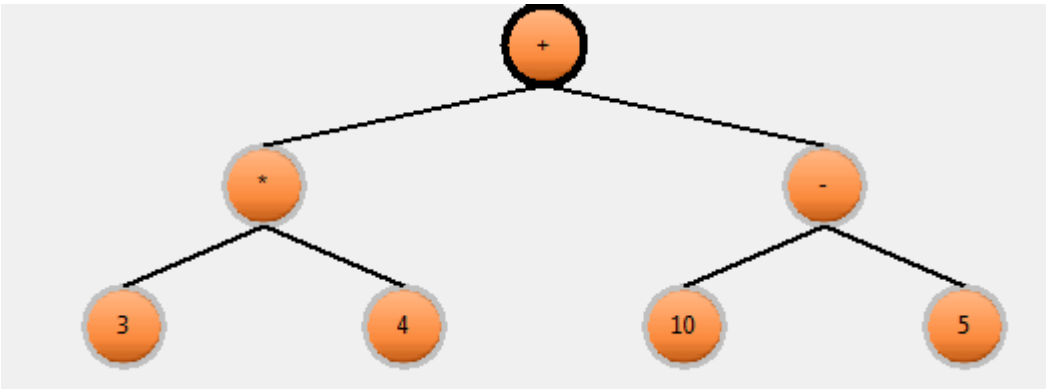

Figura 64: Instancia inicial del método evaluar (Nodo +)

En la figura 67, mostramos el resultado del recorrido “evaluar” del sub-arbol izquierdo (Por simplicidad se omitieron las instancias intermedias). Se puede observar, en el panel de instancias, el resultado de cada evaluación: “*Volviendo de la recursión * – Return(12)*”

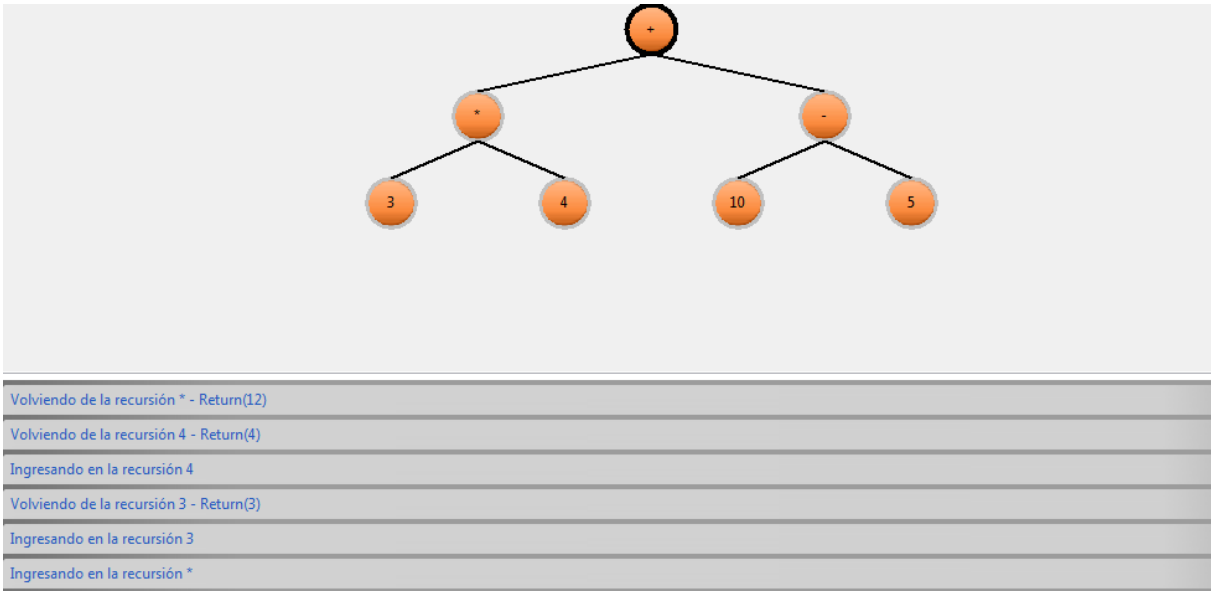

Figura 65: Árbol izquierdo procesado. (Return 12)

10.5. Ejemplo 3: Ejemplo del procesamiento de una lista recursivamente

Se mostrará un ejemplo de un recorrido de una lista de forma recursiva. Primero, configuramos la ejecución:

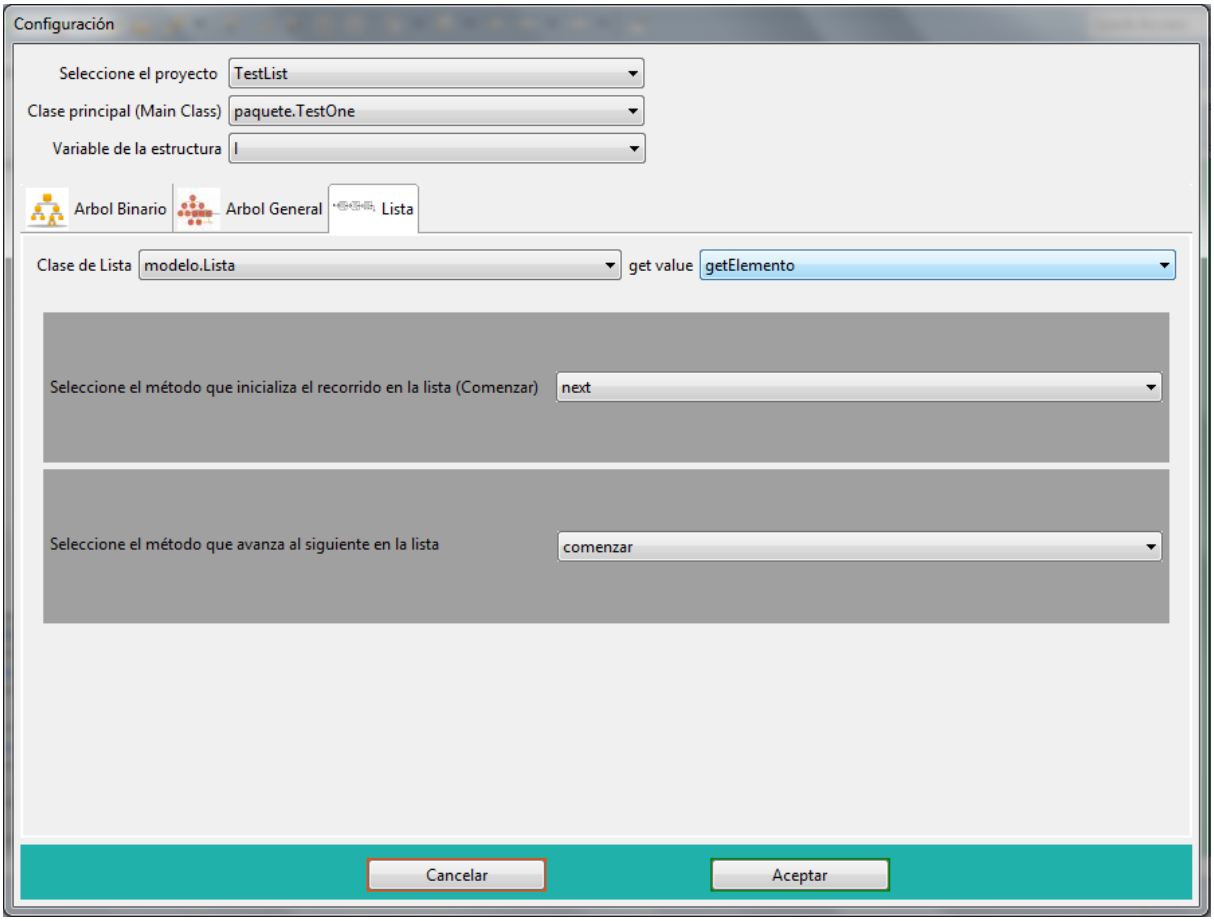

Figura 66: Ventana de configuración para estructura "Lista"

Comenzamos con la ejecución, y obtenemos el siguiente resultado:

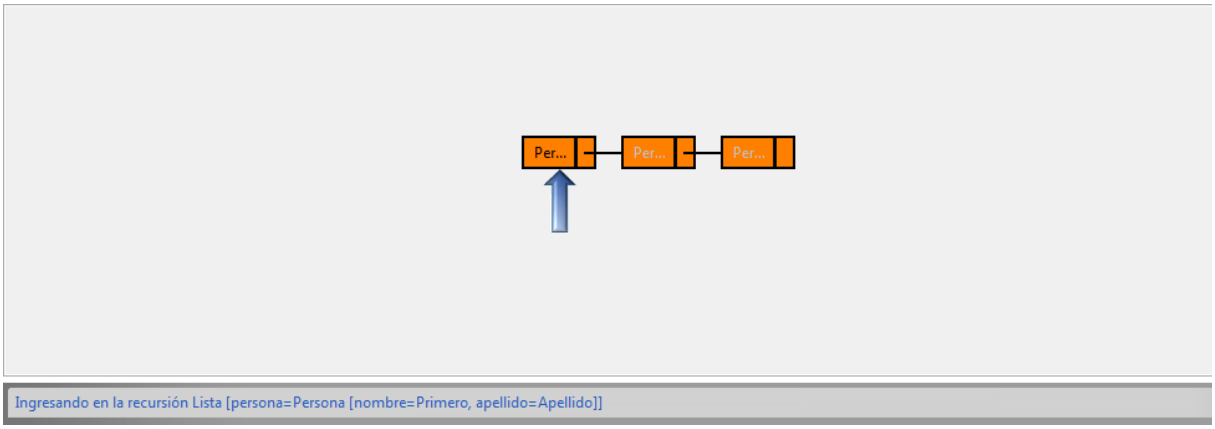

Figura 67: Instancia inicial del método "printInOrden()"

Al finalizar la ejecución, obtendremos el siguiente resultado (Nuevamente, se omitieron las instancias intermedias).

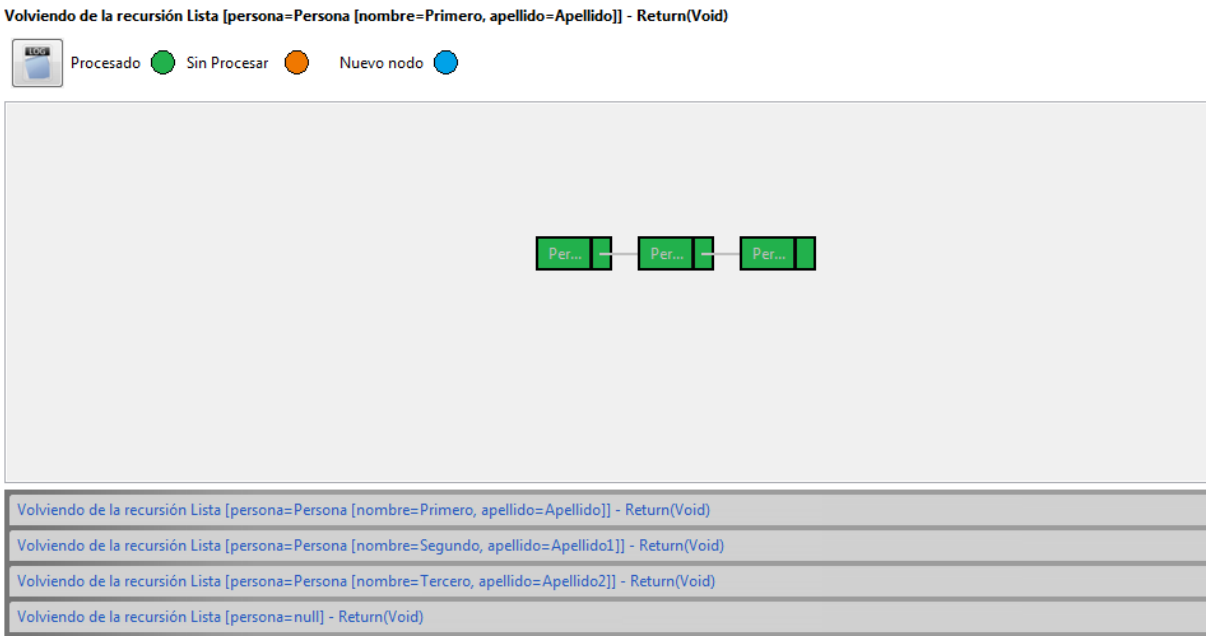

Figura 68: Ejecución finalizada del método

Capítulo 11 - Líneas de trabajo futuras y conclusiones

11.1. Líneas de trabajo futuras.

El trabajo en esta etapa se enfocó en la creación de la herramienta que interprete el código implementado por los estudiantes y lo visualice para una mejor comprensión. En esta etapa se implementó un plugin para Arboles Binarios, Arboles Generales y Listas, sin embargo, el diseño se pensó de manera genérica, posibilitando la incorporación de nuevas estructuras.

A modo de ejemplo para futuras líneas de este trabajo, describiremos los pasos que deberían seguirse para incorporar una nueva estructura como Grafos.

La Figura 69 muestra una parte del UML de nuestro diseño, donde se observa una jerarquía de transformadores los cuales se encargan, valga la redundancia, de transformar el objeto recibido en el momento que fue detenido por el aspecto. Es decir, transforma el modelo de los alumnos en una estructura similar pero propia de nuestro modelo.

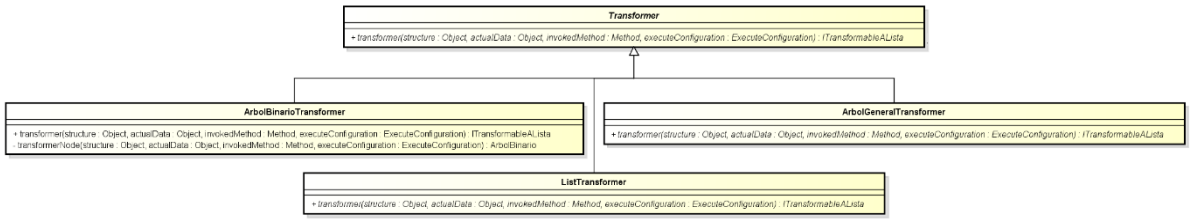

Figura 69 Jerarquía de Transformer

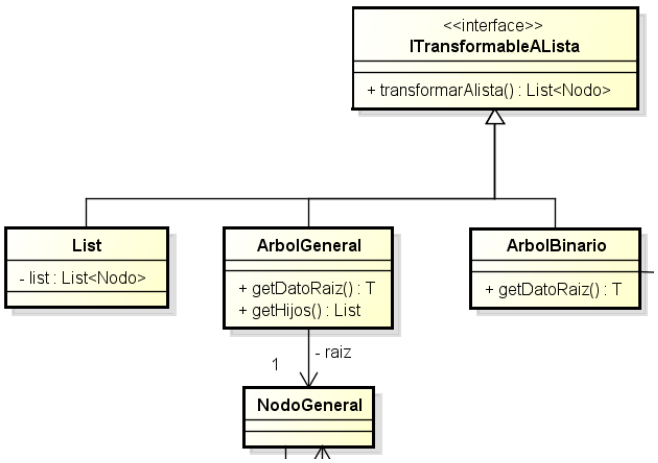

Figura 70 Jerarquía de Estructuras

Como se muestra en la imagen se debería agregar una clase “GrafoTransformer” que herede de “Transformer”.

Si prestamos atención a los siguientes fragmentos del UML, podemos observar que está implementado el patrón “Strategy” como también una jerarquía de configuraciones. La finalidad del patrón es la de crear un “Strategy” nuevo cada vez que se elija una nueva estructura en la ventana de configuración, donde a su vez creará la configuración especifica de tal estructura. De esta manera si se desea agregar la estructura de datos Grafos, se deberá crear un “GrafoConfigurationStrategy” que extienda de “StructureConfigurationStrategy” y un “ExecuteGrafoConfiguration”.

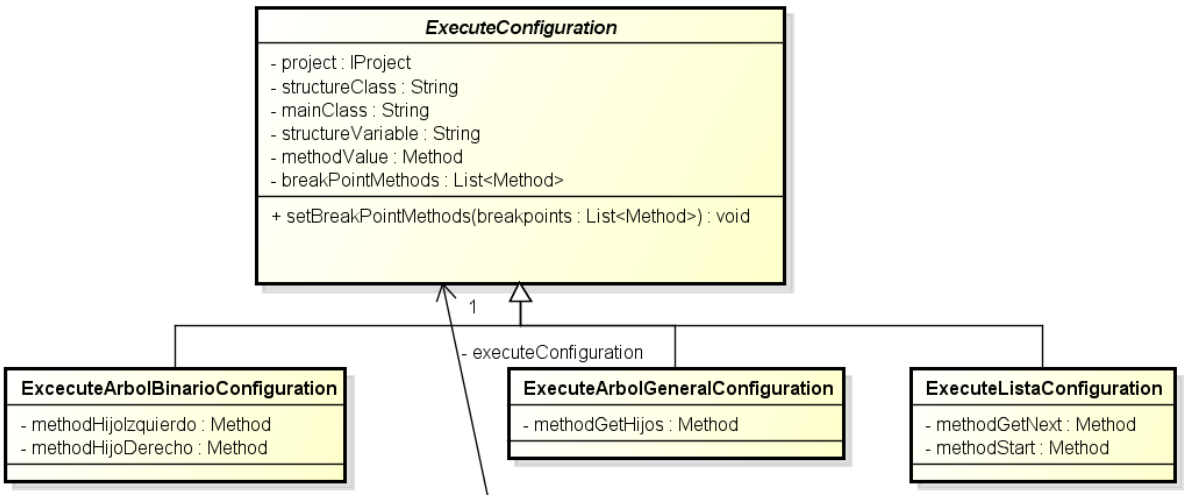

Figura 71 Jerarquia de ExecuteConfiguration

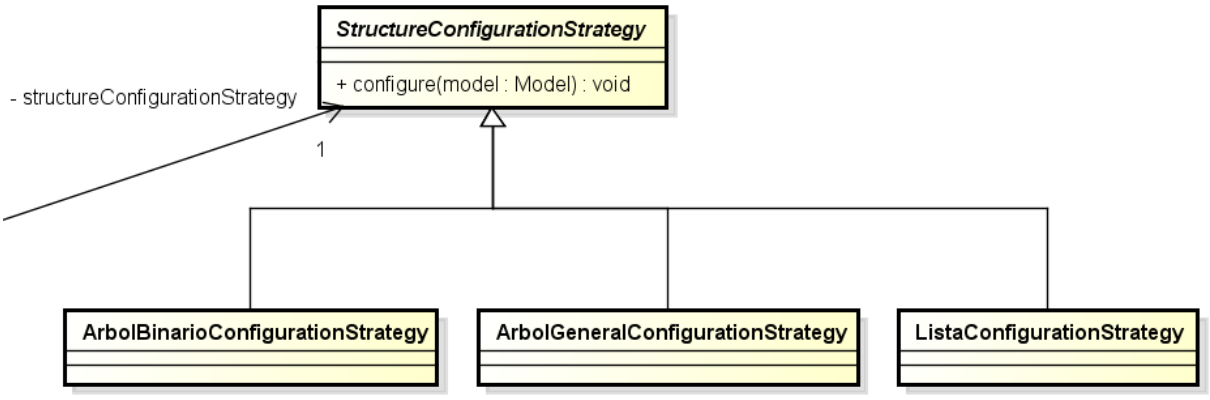

Figura 72 Jerarquía de StructureConfigurationStrategy

11.2. Conclusiones

En esta tesina se han descrito las actividades llevadas a cabo para la producción de un plugin para Eclipse que visualice estructuras de datos y sus algoritmos asociados. Para una mejor compresión de la recursión, el plugin también muestra, en caso de algoritmos recursivos, las sucesivas invocaciones.

Como parte de este trabajo, se analizaron las problemáticas con las que se encontraban los alumnos de la Facultad de Informática e Ingeniería a la hora de cursar la materia Algoritmos y Estructuras de Datos y se intentaron resolver con una solución transparente y no invasiva.

Para la implementación de la herramienta, se tuvo que investigar sobre el desarrollo de plugins para Eclipse, debido a que es el ambiente que utilizan en estas asignaturas, por lo que nos pareció apropiado brindar una solución para ese entorno. A su vez se investigó sobre Programación Orientada a Aspectos, el cual fue definido detalladamente con anterioridad explicando su uso, como también ciertas herramientas que sirvieron de apoyo para el desarrollo en cuestión.

La principal dificultad que debimos resolver, fue como “detener” la ejecución en algún punto determinado (método a inspeccionar definido por el alumno), para obtener los parámetros en tiempo de ejecución y realizar el gráfico de la estructura en ese punto. Otro de los puntos complejos que debimos afrontar, estuvo relacionado con detener el thread que se encuentra corriendo en la interfaz del usuario (Ej: loop infinito). Tal suceso no fue posible por lo que se tuvo que reestructurar parte de la arquitectura, implementando un modelo cliente servidor donde este último corre sobre un proceso; y es así como se puede detener, enviando una señal de finalización.

El desarrollo de Algolipse nos permitió profundizar nuestro conocimiento sobre JAVA, aprender Programación Orientada a Aspectos para resolver una problemática compleja vinculada a la funcionalidad del plugin y en especial aportar con nuestro desarrollo una herramienta didáctica, que esperamos sirva como complemento a las clases y prácticas brindadas por los docentes de las asignaturas mencionadas.

Capítulo 12 – Índice de imágenes

Bibliografía

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein; Introduction to Algorithms, Second Edition, ISBN 0-07-013151-1 (McGraw-Hill), 2001.

[2] D. Knuth, The Art of Computing Programming, Addison-Wesley, 1968.

[3] J. Cross, D. Hendrix; Workshop jGRASP: An Integrated Development Environment with Visualizations for Teaching Java in CS1, CS2, and Beyond, 36th ASEE/IEEE Frontiers in Education Conference, ISBN 1-4244-0256-5, 27 -31 Oct. 2006.

[4] T. Chen and T. Tarek Sobh; A Tool for Data Structure Visualization and User-defined Algorithm Animation, Department of Computer Science and engineering, University of Bridgeport, USA. Accesible en: http://www1bpt.bridgeport.edu/~risc/pdf/jp29.pdf.

[5] G. Cardim, I. Marcal, C. de Sousa, D. de Campos, C. Marin; A. do Carmo, D. Toledo, A. Saito, R. Correia, R. Garcia, Teaching and Learning of Data Structures Supported by Computer: An Experience with the CADILAG tool. Information Systems and Technologies (CISTI), 2012 7th Iberian Conference, Madrid, p:1-5. ISBN 978-1-4673-2843-2.

[6] A. Blewitt, Eclipse 4 Plug-in Development by Example: Beginner's Guide. ISBN 978-1-78216-032-8, Packt Publishing, 2013.

[7] Bruce Eckel; Thinking in Java, Fourth Edition. Prentice Hall, ISBN ISBN 0-13-187248-6, enero 2006.

[8] Colyer, Clement, Harley, Webster, Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools (1st Edition). ISBN-13: 978-0321245878.

[9] J. Gradecki, N. Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java. ISDN 0-471-43104-4, Wiley, 2003.

[10] Gradecki, Lesiecki. Mastering AspectJ: Aspect-Oriented Programming in Java (1st Edition), ISBN-13: 978-0471431046.

[11] Glen McCluskey: Using Java Reflection, enero 1998.

[12] Alex Blewit, Eclipse 4 Plug-in Development by Example Beginner's Guide How to develop, build, test, package, and release Eclipse plug-ins with features for Eclipse, Packt Publishing Ltd, ISBN 978-1-78216-032-8, junio 2013.

[13] Steve Northover; The Standard Widget Toolkit.

[14] C. Haase and R. Guy, Filthy Rich Clients: Developing Animated and Graphical Effects for Desktop Java Applications, ISBN-10: 0132413930, Addison Wesley, 2007.

[15] Matthew Scarpino, Stephen Holder, Stanford Ng y Laurent Mihalkovic: SWT/JFace in action, ISBN 1-932394-27-3, 2005.

[16] Gradecki, Lesiecki. Mastering AspectJ: Aspect-Oriented Programming in Java (1st Edition), ISBN-13: 978-0471431046.

[17] J. Cross, D. Hendrix; Workshop jGRASP: An Integrated Development Environment with Visualizations for Teaching Java in CS1, CS2, and Beyond, 36th ASEE/IEEE Frontiers in Education Conference, ISBN 1-4244-0256-5, 27 -31 Oct. 2006.

[18] T. Chen and T. Tarek Sobh; A Tool for Data Structure Visualization and User-defined Algorithm Animation, Department of Computer Science and engineering, University of Bridgeport, USA. Accesible en: http://www1bpt.bridgeport.edu/~risc/pdf/jp29.pdf.